```python
import streamlit as st
import pandas as pd
import json
import tempfile
from datetime import datetime
from typing import List, Dict, Any
from app.utils import STAParser, generate_pdf_report
from app.inference import TimingAnalyzer
from app.models import TimingPath


def setup_sidebar() -> Dict[str, Any]:
    """Setup sidebar configuration"""
    st.sidebar.header("⚙️ Configuration")

    api_key = st.sidebar.text_input(
        "Groq API Key",
        type="password",
        help="Get your API key from https://console.groq.com/"
    )

    st.sidebar.header("📁 Upload Files")
    timing_file = st.sidebar.file_uploader(
        "STA Timing Report",
        type=['txt', 'rpt', 'log'],
        help="Upload your Static Timing Analysis report"
    )

    st.sidebar.header("🔍 Analysis Options")
    analyze_violations_only = st.sidebar.checkbox(
        "Analyze violations only",
        value=True,
        help="Only analyze paths with timing violations"
    )

    show_raw_data = st.sidebar.checkbox(
        "Show raw parsed data",
        value=False
```

```
    )

    return {
        "api_key": api_key,
        "timing_file": timing_file,
        "analyze_violations_only":
analyze_violations_only,
        "show_raw_data": show_raw_data
    }


def display_analysis_results(analyses: List[Dict],
config: Dict):
    """Display analysis results in an interactive
format"""
    st.header("🔍 AI Analysis Results")

    # Summary statistics
    total = len(analyses)
    violated = sum(1 for a in analyses if a.get('
status') == 'VIOLATED')
    met = total - violated

    col1, col2, col3 = st.columns(3)
    with col1:
        st.metric("Total Paths", total)
    with col2:
        st.metric("Violations", violated, delta=f"-{
violated}" if violated else None)
    with col3:
        st.metric("Met Timing", met)

    # Detailed analysis for each path
    for i, analysis in enumerate(analyses, 1):
        with st.expander(f"Path {i}: {analysis.get('
startpoint')} → {analysis.get('endpoint')}"):
            col1, col2 = st.columns([1, 2])

            with col1:
                st.subheader("Path Details")
                st.info(f"**Type:** {analysis.get('
```

```
 73 path_type')}")
 74                 status = analysis.get('status')
 75                 if status == "VIOLATED":
 76                     st.error(f"**Status:** {status}
     (Slack: {analysis.get('slack')} ns)")
 77                     st.error(f"**Severity:** {
    analysis.get('severity', 'unknown')}")
 78                 else:
 79                     st.success(f"**Status:** {status
    }")
 80
 81                 st.write(f"**Startpoint:** {analysis
    .get('startpoint')}")
 82                 st.write(f"**Endpoint:** {analysis.
    get('endpoint')}")
 83
 84             with col2:
 85                 st.subheader("Technical Analysis")
 86                 if status == "VIOLATED":
 87                     st.write(f"**Root Cause:** {
    analysis.get('root_cause')}")
 88                     st.write(f"**Estimated Effort
    :** {analysis.get('estimated_effort')}")
 89
 90                     st.subheader("Recommended Fixes"
    )
 91                     suggestions = analysis.get('
    suggestions', [])
 92                     for j, suggestion in enumerate(
    suggestions, 1):
 93                         priority = suggestion.get('
    priority', '').upper()
 94                         priority_color = {
 95                             'HIGH': 'red',
 96                             'MEDIUM': 'orange',
 97                             'LOW': 'green'
 98                         }.get(priority, 'gray')
 99
100                         st.markdown(
101                             f"**{j}. {suggestion.get
    ('fix')}** "
```

```
                                f"<span style='color:{priority_color}'>[{priority}]</span>",
                                unsafe_allow_html=True
                            )
                            st.caption(f"*{suggestion.get('explanation')}*")
                else:
                    st.success("✅ Timing requirements met successfully")


def create_download_buttons(analyses: List[Dict], parsed_paths: List[TimingPath]):
    """Create download buttons for analysis results"""
    col1, col2 = st.columns(2)

    with col1:
        # JSON download
        json_data = json.dumps({
            "timestamp": datetime.now().isoformat(),
            "analyses": analyses,
            "original_paths": [p.dict() for p in parsed_paths]
        }, indent=2)

        st.download_button(
            label="📥 Download JSON Report",
            data=json_data,
            file_name="timing_analysis.json",
            mime="application/json"
        )

    with col2:
        # PDF download
        if st.button("📄 Generate PDF Report"):
            with tempfile.NamedTemporaryFile(suffix=".pdf", delete=False) as tmp:
                pdf_path = generate_pdf_report(analyses, tmp.name)
                with open(pdf_path, "rb") as f:
```

```
                    st.download_button(
                        label="⬇️ Download PDF Report",
                        data=f.read(),
                        file_name="timing_analysis_report.pdf",
                        mime="application/pdf"
                    )


def show_instructions():
    """Show usage instructions"""
    st.info("""
    ## 🚀 How to Use This Tool

    1. **Get API Key**: Sign up at [Groq Console](https://console.groq.com/) for free access
    2. **Upload Report**: Upload your STA timing report (.txt, .rpt, .log)
    3. **Configure**: Choose analysis options in the sidebar
    4. **Analyze**: Click the 'Run Analysis' button
    5. **Review**: Examine AI-powered insights and recommendations
    6. **Export**: Download detailed reports in JSON or PDF format

    ## 📋 Supported Formats

    The parser works with standard STA report formats from tools like:
    - Synopsys PrimeTime
    - Cadence Tempus
    - Siemens Questa
    - OpenSTA
    """)


def main_ui():
    """Main UI function"""
    config = setup_sidebar()
```

```
168
169     if config["timing_file"] and config["api_key"]:
170         if st.button("🚀 Run Analysis", type="primary
    "):
171             with st.spinner("Parsing timing report
    ..."):
172                 report_content = config["timing_file
    "].getvalue().decode("utf-8")
173                 parser = STAParser(report_content)
174                 parsed_paths = parser.parse()
175
176             if not parsed_paths:
177                 st.warning("No valid timing paths
    found in the report")
178                 return
179
180             # Filter paths if needed
181             if config["analyze_violations_only"]:
182                 paths_to_analyze = [p for p in
    parsed_paths if p.status == "VIOLATED"]
183                 st.info(f"Analyzing {len(
    paths_to_analyze)} violated paths (of {len(
    parsed_paths)} total)")
184             else:
185                 paths_to_analyze = parsed_paths
186
187             # Show raw data if requested
188             if config["show_raw_data"]:
189                 with st.expander("📄 Raw Parsed Data"
    ):
190                     st.json([p.dict() for p in
    parsed_paths])
191
192             # Run analysis
193             analyzer = TimingAnalyzer(config["
    api_key"])
194             analyses = analyzer.analyze_paths(
    paths_to_analyze)
195
196             # Display results
197             display_analysis_results(analyses,
```

```
197 config)
198             create_download_buttons(analyses,
    parsed_paths)
199
200     else:
201         show_instructions()
```

```python
import re
import json
import tempfile
from typing import List, Dict, Any, Optional
from datetime import datetime
from reportlab.lib.pagesizes import letter
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Table, TableStyle
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib import colors
from app.models import TimingPath


class STAParser:
    def __init__(self, sta_report: str):
        self.report = sta_report

    def parse(self) -> List[TimingPath]:
        paths = []
        blocks = self.report.strip().split("Startpoint:")

        for block in blocks[1:]:
            path_data = self._parse_block(block)
            if path_data:
                try:
                    paths.append(TimingPath(**path_data))
                except Exception as e:
                    print(f"Error parsing block: {e}")
                    continue

        return paths

    def _parse_block(self, block: str) -> Optional[Dict[str, Any]]:
        lines = block.strip().splitlines()
        if not lines:
            return None
```

```
36
37          startpoint = lines[0].strip()
38          endpoint = ""
39          clock = ""
40          path_type = ""
41          data_arrival = None
42          data_required = None
43          slack = None
44          status = "MET"
45          logic_chain = []
46
47          for line in lines:
48              line = line.strip()
49              if line.startswith("Endpoint:"):
50                  endpoint = line.replace("Endpoint:",
    "").strip()
51              elif line.startswith("Path Group:"):
52                  clock = line.replace("Path Group:",
    "").strip()
53              elif line.startswith("Path Type:"):
54                  path_type = line.replace("Path Type:"
    , "").strip()
55              elif "data arrival time" in line and
    data_arrival is None:
56                  try:
57                      data_arrival = float(line.split
    ()[0])
58                  except (ValueError, IndexError):
59                      pass
60              elif "data required time" in line and
    data_required is None:
61                  try:
62                      data_required = float(line.split
    ()[0])
63                  except (ValueError, IndexError):
64                      pass
65              elif "slack" in line.lower():
66                  parts = line.split()
67                  try:
68                      slack = float(parts[0])
69                      if "violated" in line.lower():
```

```
                        status = "VIOLATED"
                except (ValueError, IndexError):
                    pass
            elif re.search(r"\s+[0-9]", line) and ("v " in line or "^ " in line):
                parts = line.split()
                if len(parts) >= 3:
                    try:
                        delay = float(parts[0])
                    except ValueError:
                        delay = 0.0
                    description = " ".join(parts[2:])
                    logic_chain.append({"cell": description, "delay": delay})

        return {
            "startpoint": startpoint,
            "endpoint": endpoint,
            "clock": clock,
            "path_type": path_type,
            "data_arrival_time": data_arrival,
            "data_required_time": data_required,
            "slack": slack,
            "status": status,
            "logic_chain": logic_chain
        }


def generate_pdf_report(analyses: List[Dict], output_path: str):
    """Generate PDF report from analysis results"""
    doc = SimpleDocTemplate(output_path, pagesize=letter)
    styles = getSampleStyleSheet()
    story = []

    # Title
    title_style = ParagraphStyle(
        'Title',
        parent=styles['Heading1'],
```

```
106         fontSize=16,
107         spaceAfter=30
108     )
109     story.append(Paragraph("Timing Violation
    Analysis Report", title_style))
110     story.append(Spacer(1, 12))
111 
112     # Summary
113     violated = sum(1 for a in analyses if a.get('
    status') == 'VIOLATED')
114     story.append(Paragraph(f"Total Paths Analyzed: {
    len(analyses)}", styles['Normal']))
115     story.append(Paragraph(f"Violated Paths: {
    violated}", styles['Normal']))
116     story.append(Spacer(1, 20))
117 
118     # Detailed analysis
119     for i, analysis in enumerate(analyses, 1):
120         if analysis.get('status') == 'VIOLATED':
121             story.append(Paragraph(f"Violation {i}:
    {analysis.get('startpoint')} → {analysis.get('
    endpoint')}",
122                                    styles['Heading2'
    ]))
123 
124             data = [
125                 ["Slack", f"{analysis.get('slack', '
    N/A')} ns"],
126                 ["Path Type", analysis.get('
    path_type', 'N/A')],
127                 ["Severity", analysis.get('severity'
    , 'N/A')],
128                 ["Root Cause", analysis.get('
    root_cause', 'N/A')]
129             ]
130 
131             table = Table(data, colWidths=[100, 400
    ])
132             table.setStyle(TableStyle([
133                 ('BACKGROUND', (0, 0), (-1, 0),
    colors.grey),
```

```
134                 ('TEXTCOLOR', (0, 0), (-1, 0),
    colors.whitesmoke),
135                 ('ALIGN', (0, 0), (-1, -1), 'LEFT'),
136                 ('FONTNAME', (0, 0), (-1, 0), '
    Helvetica-Bold'),
137                 ('FONTSIZE', (0, 0), (-1, -1), 10),
138                 ('BOTTOMPADDING', (0, 0), (-1, 0),
    12),
139                 ('BACKGROUND', (0, 1), (-1, -1),
    colors.beige),
140                 ('GRID', (0, 0), (-1, -1), 1, colors
    .black)
141             ]))
142 
143             story.append(table)
144             story.append(Spacer(1, 12))
145 
146             # Suggestions
147             story.append(Paragraph("Recommended
    Fixes:", styles['Heading3']))
148             for suggestion in analysis.get('
    suggestions', []):
149                 story.append(
150                     Paragraph(f"• {suggestion.get('
    fix')} ({suggestion.get('priority')} priority)",
    styles['Normal']))
151                 story.append(Paragraph(f"
    Explanation: {suggestion.get('explanation')}",
    styles['Italic']))
152 
153             story.append(Spacer(1, 20))
154 
155     doc.build(story)
156     return output_path
```

```python
from pydantic import BaseModel, Field
from typing import List, Optional, Dict, Any

class TimingPath(BaseModel):
    startpoint: str
    endpoint: str
    clock: str
    path_type: str
    data_arrival_time: Optional[float] = None
    data_required_time: Optional[float] = None
    slack: Optional[float] = None
    status: str
    logic_chain: List[Dict[str, Any]]

class AnalysisSuggestion(BaseModel):
    fix: str
    priority: str  # high, medium, low
    explanation: str

class ViolationAnalysis(BaseModel):
    startpoint: str
    endpoint: str
    path_type: str
    status: str
    slack: Optional[float] = None
    root_cause: Optional[str] = None
    severity: Optional[str] = None
    suggestions: Optional[List[AnalysisSuggestion
]] = None
    estimated_effort: Optional[str] = None

class AnalysisReport(BaseModel):
    timestamp: str
    total_paths: int
    violated_paths: int
    analyses: List[ViolationAnalysis]
    summary: Dict[str, Any]
```

```
PROMPT_TEMPLATE = """
**ROLE**: You are a Senior Timing Analysis Engineer with 15+ years of experience at a leading semiconductor foundry. Your expertise is in timing closure at advanced nodes (7nm, 5nm, 3nm).

**TASK**: Perform quantitative timing violation analysis and provide specific, actionable recommendations with exact numbers.

**QUANTITATIVE ANALYSIS FRAMEWORK**:
1. Calculate total path delay from logic_chain delays
2. Identify the specific cell contributing most to delay (% contribution)
3. Relate slack violation to clock period quantitatively
4. Provide exact numbers for expected improvements

**INPUT PATH JSON**:
{path_json}

**STRICT INSTRUCTIONS**:
1. For VIOLATED paths:
   - Root cause MUST include: exact delay numbers, worst-cell identification, quantitative analysis
   - Suggestions MUST be specific: cell names, library types, exact expected improvement
   - Use semiconductor engineering terminology

2. For MET paths: Simply return {{"status": "MET"}} to conserve tokens

**REQUIRED OUTPUT FORMAT** (ONLY JSON):
{{
  "startpoint": "string",
  "endpoint": "string",
  "path_type": "string",
  "status": "string",
  "slack": float,
  "quantitative_analysis": {{
    "total_path_delay": float,
```

```
    "clock_period": float,
    "required_delay_reduction": float,
    "worst_cell": {{
      "name": "string",
      "delay_contribution": float,
      "percentage": float
    }}
  }},
  "root_cause": "string (with numbers)",
  "severity": "critical|high|medium|low",
  "specific_fixes": [
    {{
      "fix": "string (specific action)",
      "priority": "high|medium|low",
      "expected_improvement": "float ns",
      "implementation_command": "string (tool-
specific)",
      "risk": "string"
    }}
  ],
  "estimated_engineering_effort": "hours"
}}

**QUANTITATIVE EXAMPLE**:
{{
  "startpoint": "U1/Q",
  "endpoint": "U5/D",
  "path_type": "max",
  "status": "VIOLATED",
  "slack": -0.85,
  "quantitative_analysis": {{
    "total_path_delay": 5.20,
    "clock_period": 4.35,
    "required_delay_reduction": 0.85,
    "worst_cell": {{
      "name": "XOR2X1",
      "delay_contribution": 1.8,
      "percentage": 34.6
    }}
  }},
  "root_cause": "Setup violation: Path delay 5.20ns
```

```
exceeds clock period 4.35ns by 0.85ns. Main contributor: XOR2X1 (1.8ns, 34.6% of total delay)",
  "severity": "high",
  "specific_fixes": [
    {{
      "fix": "Replace XOR2X1 with XOR2X4",
      "priority": "high",
      "expected_improvement": "0.3-0.4ns",
      "implementation_command": "replace_cell -cell XOR2X1 -with XOR2X4 -area_penalty",
      "risk": "Low: +5% area, minimal power impact"
    }},
    {{
      "fix": "Increase drive strength of AND2X1 to AND2X2",
      "priority": "medium",
      "expected_improvement": "0.15ns",
      "implementation_command": "size_cell -cell AND2X1 -to AND2X2",
      "risk": "None"
    }}
  ],
  "estimated_engineering_effort": 2
}}
"""

FEW_SHOT_EXAMPLE = [
    {
        "input": {
            "startpoint": "reg1/Q",
            "endpoint": "reg2/D",
            "slack": -1.2,
            "path_type": "max",
            "data_arrival_time": 6.2,
            "data_required_time": 5.0,
            "logic_chain": [
                {"cell": "BUFX4", "delay": 0.4},
                {"cell": "NAND3X2", "delay": 0.9},
                {"cell": "OR2X1", "delay": 0.5},
                {"cell": "XOR2X1", "delay": 1.2}
            ]
```

```
        },
        "output": {
            "quantitative_analysis": {
                "total_path_delay": 6.2,
                "clock_period": 5.0,
                "required_delay_reduction": 1.2,
                "worst_cell": {
                    "name": "XOR2X1",
                    "delay_contribution": 1.2,
                    "percentage": 41.4
                }
            },
            "root_cause": "Setup violation: Path delay 6.2ns exceeds clock period 5.0ns by 1.2ns. Primary bottleneck: XOR2X1 contributes 1.2ns (41.4% of total delay) due to high output load",
            "severity": "critical",
            "specific_fixes": [
                {
                    "fix": "Replace XOR2X1 with XOR2X4 and add buffer tree",
                    "priority": "high",
                    "expected_improvement": "0.6-0.8ns",
                    "implementation_command": "replace_cell [get_cells XOR2X1] XOR2X4; insert_buffer -net [get_nets -of [get_pins XOR2X1/Z]] -distance 20",
                    "risk": "Medium: +15% area, routing congestion possible"
                },
                {
                    "fix": "Clone NAND3X2 to reduce fanout load on XOR2X1",
                    "priority": "medium",
                    "expected_improvement": "0.3ns",
                    "implementation_command": "clone_cell [get_cells NAND3X2] NAND3X2_CLONE; connect_pins appropriately",
                    "risk": "Low: minimal area impact"
```

```
                }
            ],
            "estimated_engineering_effort": 4
        }
    }
]
```

```python
import os
import json
from typing import List, Dict, Any
from langchain.chat_models import init_chat_model
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import JsonOutputParser
from app.constants import PROMPT_TEMPLATE
from app.models import TimingPath


class TimingAnalyzer:
    def __init__(self, api_key: str):
        self.api_key = api_key
        self.model = self._initialize_model()
        self.json_parser = JsonOutputParser()

    def _initialize_model(self):
        """Initialize the Groq model"""
        os.environ["GROQ_API_KEY"] = self.api_key
        return init_chat_model(
            "llama-3.3-70b-versatile",
            model_provider="groq",
            temperature=0.1
        )

    def analyze_paths(self, paths: List[TimingPath]) -> List[Dict[str, Any]]:
        """Analyze timing paths using LLM"""
        results = []
        prompt_template = PromptTemplate.from_template(PROMPT_TEMPLATE)

        for i, path in enumerate(paths):
            try:
                chain = prompt_template | self.model | self.json_parser
                result = chain.invoke({"path_json": json.dumps(path.dict(), indent=2)})

                # Ensure result has required fields
```

```
                    result.update({
                        "startpoint": path.startpoint,
                        "endpoint": path.endpoint,
                        "path_type": path.path_type,
                        "status": path.status,
                        "slack": path.slack
                    })

                    results.append(result)

                except Exception as e:
                    print(f"Error analyzing path {i}: {e}
")
                    # Create a basic result for failed
analysis
                    results.append({
                        "startpoint": path.startpoint,
                        "endpoint": path.endpoint,
                        "path_type": path.path_type,
                        "status": path.status,
                        "slack": path.slack,
                        "root_cause": f"Analysis failed:
{str(e)}",
                        "severity": "unknown",
                        "suggestions": [],
                        "estimated_effort": "unknown"
                    })

            return results
```

```
{
  "paths": [
    {
      "startpoint": "in1 (input port clocked by clk)",
      "endpoint": "r1 (rising edge-triggered flip-flop clocked by clk)",
      "clock": "clk",
      "path_type": "min",
      "data_arrival_time": 0.0,
      "data_required_time": 0.0,
      "slack": 0.0,
      "status": "VIOLATED",
      "logic_chain": [
        {
          "cell": "v input external delay",
          "delay": 0.0
        },
        {
          "cell": "v in1 (in)",
          "delay": 0.0
        },
        {
          "cell": "v r1/D (DFF_X1)",
          "delay": 0.0
        },
        {
          "cell": "r1/CK (DFF_X1)",
          "delay": 0.0
        }
      ]
    },
    {
      "startpoint": "r2 (rising edge-triggered flip-flop clocked by clk)",
      "endpoint": "r3 (rising edge-triggered flip-flop clocked by clk)",
      "clock": "clk",
      "path_type": "max",
      "data_arrival_time": 0.41,
      "data_required_time": 9.84,
```

```
      "slack": 9.43,
      "status": "MET",
      "logic_chain": [
        "^ r2/CK (DFF_X1)",
        "v r2/Q (DFF_X1)",
        "v u1/Z (BUF_X1)",
        "v u2/ZN (AND2_X1)",
        "v r3/D (DFF_X1)",
        "r3/CK (DFF_X1)"
      ]
    }
  ]
}
```

```
#%%
import re
import json

class STAParser:
    def __init__(self, sta_report: str):
        self.report = sta_report

    def parse(self):
        paths = []
        blocks = self.report.strip().split("Startpoint:")
        for block in blocks[1:]:
            path = self._parse_block(block)
            if path:
                paths.append(path)
        return {"paths": paths}

    def _parse_block(self, block: str):
        lines = block.strip().splitlines()
        startpoint = lines[0].strip()
        endpoint = ""
        clock = ""
        path_type = ""
        data_arrival = None
        data_required = None
        slack = None
        status = None
        logic_chain = []

        # Extract metadata
        for line in lines:
            if line.startswith("Endpoint:"):
                endpoint = line.replace("Endpoint:", "").strip()
            elif line.startswith("Path Group:"):
                clock = line.replace("Path Group:", "").strip()
            elif line.startswith("Path Type:"):
                path_type = line.replace("Path Type:", "").strip()
```

```
            elif "data arrival time" in line and data_arrival is None:
                try:
                    data_arrival = float(line.split()[0])
                except:
                    pass
            elif "data required time" in line and data_required is None:
                try:
                    data_required = float(line.split()[0])
                except:
                    pass
            elif "slack" in line:
                parts = line.split()
                slack = float(parts[0])
                status = "VIOLATED" if "VIOLATED" in line else "MET"
            elif re.search(r"\s+[0-9]", line) and ("v " in line or "^ " in line):
                # Logic chain row
                parts = line.split()
                try:
                    delay = float(parts[0])
                except:
                    delay = 0.0
                description = " ".join(parts[2:])
                logic_chain.append({"cell": description, "delay": delay})

        # Compact JSON output
        if status == "VIOLATED":
            return {
                "startpoint": startpoint,
                "endpoint": endpoint,
                "clock": clock,
                "path_type": path_type,
                "data_arrival_time": data_arrival,
                "data_required_time": data_required,
                "slack": slack,
```

```
                "status": status,
                "logic_chain": logic_chain
            }
        else:  # summary mode
            return {
                "startpoint": startpoint,
                "endpoint": endpoint,
                "clock": clock,
                "path_type": path_type,
                "data_arrival_time": data_arrival,
                "data_required_time": data_required,
                "slack": slack,
                "status": status,
                "logic_chain": [x["cell"] for x in
logic_chain]
            }


if __name__ == "__main__":
    with open(r"D:\Code\timing-violaton-debugger\app
\data\timing_report.txt", "r") as f:
        report = f.read()

    parser = STAParser(report)
    output = parser.parse()

    with open("sta_output.json", "w") as f:
        json.dump(output, f, indent=2)

#%%
import os
import json
from dotenv import load_dotenv

from langchain.chat_models import init_chat_model
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import
JsonOutputParser

# Load API key from .env
load_dotenv()
```

```
110 if not os.getenv("GROQ_API_KEY"):
111     raise ValueError("Missing GROQ_API_KEY in .env")
112 
113 # Init Groq model (can swap model easily)
114 model = init_chat_model(
115     "llama-3.3-70b-versatile",
116     model_provider="groq",
117     temperature=0.2  # lower temp for deterministic
    outputs
118 )
119 ____________________________________________________
120 #%%
121 with open("sta_output.json", "r") as f:
122     data = json.load(f)
123 
124 paths = data.get("paths", [])  # our optimized
    Module-2 output
125 print(f"Loaded {len(paths)} timing paths.")
126 ____________________________________________________
127 #%%
128 # Few-shot example (curly braces doubled)
129 few_shot_examples = """
130 ### EXAMPLE
131 Input:
132 {{
133   "startpoint": "U1/Q",
134   "endpoint": "U5/D",
135   "slack": -0.85,
136   "delay": 5.2,
137   "clock": "clk_main"
138 }}
139 
140 Output:
141 {{
142   "root_cause": "Path delay exceeds the clock period
    , causing setup violation.",
143   "suggestions": [
144     "Insert a pipeline register to break the path.",
145     "Optimize combinational logic to reduce delay.",
146     "Replace slow gates with faster cells."
147   ]
```

```
148 }}
149 """
150 
151 prompt_violation_debugger = PromptTemplate.
    from_template(
152     f"""
153 You are a GenAI-powered timing violation debugger.
154 You will receive one STA path at a time in JSON
    format.
155 
156 ### Input Path JSON:
157 {{path_json}}
158 
159 ### INSTRUCTION:
160 1. If status = "VIOLATED":
161    * Identify the root cause based on path_type and
    logic_chain.
162    * Suggest an optimal fix (1-2 sentences).
163 2. If status = "MET":
164    * root_cause = null
165    * suggested_fix = null
166 
167 ### OUTPUT:
168 Return ONLY valid JSON with these keys:
169 "startpoint", "endpoint", "path_type", "status", "
    root_cause", "suggested_fix".
170 
171 {few_shot_examples}
172 """
173 )
174 
175 # JSON parser
176 json_parser = JsonOutputParser()
177 
178 #%%
179 results = []
180 
181 for i, p in enumerate(paths, start=1):
182     # Chain: prompt + model
183     chain = prompt_violation_debugger | model
184 
```

```
185     # Invoke LLM
186     res = chain.invoke({"path_json": json.dumps(p,
    indent=2)})
187 
188     # Parse output
189     parsed_res = json_parser.parse(res.content)
190     results.append(parsed_res)
191 
192 print("Sample result:")
193 print(json.dumps(results[:1], indent=2))
194 
195 #%%
196 print(json.dumps(results, indent=2))
197 #%%
198 output = {"analysis": results}
199 
200 with open("llm_analysis.json", "w") as f:
201     json.dump(output, f, indent=2)
202 
203 print("Saved -> llm_analysis.json")
204 
205 #%%
206 
```

```json
{
  "analysis": [
    {
      "startpoint": "in1 (input port clocked by clk)",
      "endpoint": "r1 (rising edge-triggered flip-flop clocked by clk)",
      "path_type": "min",
      "status": "VIOLATED",
      "root_cause": "The minimum path delay is not meeting the required timing, likely due to insufficient delay in the logic chain or issues with clock synchronization.",
      "suggested_fix": "Review the logic chain for opportunities to insert delays or optimize the combinational logic to improve timing, and verify clock tree synchronization."
    },
    {
      "startpoint": "r2 (rising edge-triggered flip-flop clocked by clk)",
      "endpoint": "r3 (rising edge-triggered flip-flop clocked by clk)",
      "path_type": "max",
      "status": "MET",
      "root_cause": null,
      "suggested_fix": null
    }
  ]
}
```