

# Project 2: Web Content Analyzer - Key Concepts

|   |          |
|---|----------|
| <b>Project 2: Web Content Analyzer - Key Concepts</b> | <b>1</b> |
| 1. Web Scraping and Content Extraction                | 1        |
| 2. Content Processing Pipeline                        | 4        |
| 3. Multi-Layer Architecture with API Separation       | 8        |
| 4. Security in Web Scraping (SSRF Prevention)         | 12       |
| 5. Structured Content Analysis with LLMs              | 16       |
| 6. Advanced Streamlit UI Components                   | 20       |

## 1. Web Scraping and Content Extraction

### Concept Overview

Web scraping is the automated process of extracting data from websites, transforming unstructured HTML into structured, usable data for analysis and processing.

### Problem It Solves

The web contains vast amounts of valuable information that is often not available through APIs or structured formats, making it difficult to automatically analyze and process content across multiple sources.

## Solution Approach

- HTML Parsing: Using BeautifulSoup to navigate and extract data from HTML documents
- Request Management: Making HTTP requests with proper headers, timeouts, and error handling
- Content Selection: Using CSS selectors and XPath to target specific elements
- Anti-Detection Measures: Implementing user-agent rotation and request delays to prevent blocking

## Implementation Insight

Python

```
import requests
from bs4 import BeautifulSoup
import random
import time
from typing import Dict, Any, Optional

class WebScraper:
    def __init__(self):
        self.session = requests.Session()
        self.user_agents = [
            "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36",
            "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15",
            # Additional user agents...
        ]

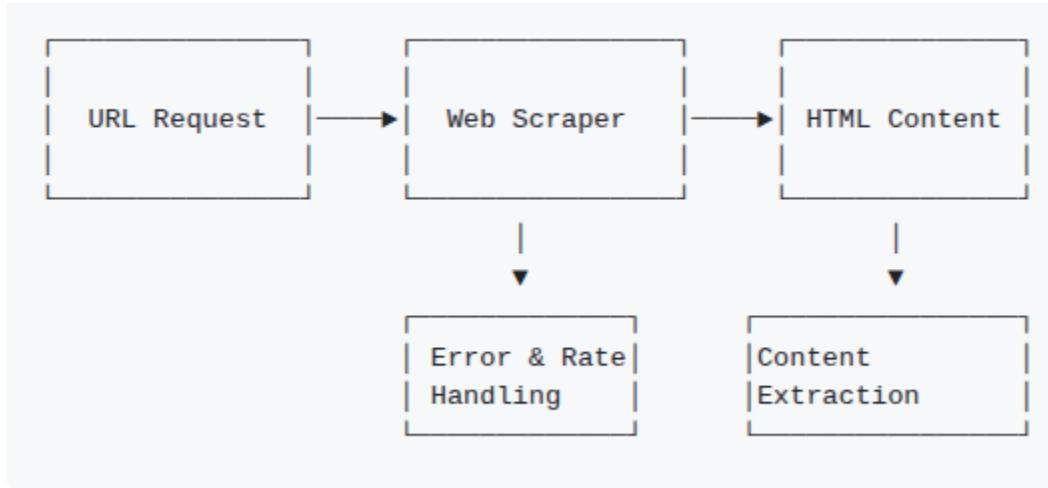
    def _get_headers(self) -> Dict[str, str]:
        """Generate request headers with random user agent"""
        return {
            "User-Agent": random.choice(self.user_agents),
            "Accept": "text/html,application/xhtml+xml,application/xml",
            "Accept-Language": "en-US,en;q=0.9"
        }
```

```
def scrape_url(self, url: str) -> Optional[BeautifulSoup]:  
    """Fetch and parse a webpage"""  
    try:  
        headers = self._get_headers()  
        response = self.session.get(url, headers=headers, timeout=10)  
        response.raise_for_status()  
  
        # Introduce a small delay to be respectful  
        time.sleep(random.uniform(0.5, 1.5))  
  
        # Parse the HTML  
        soup = BeautifulSoup(response.text, 'html.parser')  
        return soup  
  
    except requests.exceptions.RequestException as e:  
        logger.error(f"Scraping error for {url}: {str(e)}")  
        return None
```

## Common Pitfalls

- Ignoring Robots.txt: Failing to respect website crawling policies, potentially leading to IP bans or legal issues
- Inadequate Error Handling: Not properly handling network timeouts, connection errors, or unexpected HTML structures
- No Rate Limiting: Sending too many requests too quickly, triggering anti-bot measures
- Overreliance on CSS Selectors: Creating fragile scrapers that break when website designs change
- Missing Content Validation: Not verifying that extracted content is complete and meaningful

## Architecture Diagram



## Further Resources

- [BeautifulSoup Documentation](#)
- [Requests Library Documentation](#)
- [Web Scraping Ethics & Best Practices](#)
- [Mozilla Developer Network: Web Scraping Guide](#)
- [Web Scraping Legal Guide](#)

## 2. Content Processing Pipeline

### Concept Overview

Content processing pipelines transform raw extracted web content into clean, structured data through a series of filtering, cleaning, and normalization steps.

### Problem It Solves

Raw content from websites often contains irrelevant elements (ads, navigation, footers), inconsistent formatting, and excessive noise that must be filtered before meaningful analysis can occur.

### Solution Approach

- Main Content Detection: Identifying and extracting the primary content from a webpage
- Noise Removal: Eliminating boilerplate, navigation, advertisements, and other irrelevant content
- Text Normalization: Standardizing text formats, handling special characters, and removing redundant whitespace
- Structural Preservation: Maintaining document structure (headings, paragraphs) for context-aware analysis

## Implementation Insight

Python

```
class ContentProcessor:  
    def __init__(self):  
        # Common patterns for noise elements  
        self.noise_selectors = [  
            "nav", "header", "footer", "aside",  
            ".advertisement", ".sidebar", "#comments",  
            "[class*='cookie']", "[class*='banner']"  
        ]  
  
        def extract_main_content(self, soup: BeautifulSoup) -> Dict[str, Any]:  
            """Extract the main content from a webpage"""  
            # Remove noise elements  
            for selector in self.noise_selectors:  
                for element in soup.select(selector):  
                    element.decompose()  
  
            # Extract key elements  
            result = {  
                "title": self._extract_title(soup),  
                "headings": self._extract_headings(soup),  
                "main_text": self._extract_main_text(soup),  
                "links": self._extract_links(soup)  
            }  
    }
```

```
return result

def _extract_title(self, soup: BeautifulSoup) -> str:
    """Extract the page title"""
    title_tag = soup.find("title")
    if title_tag:
        return self._clean_text(title_tag.text)
    return ""

def _extract_main_text(self, soup: BeautifulSoup) -> str:
    """Extract the main article content using heuristics"""
    # Article tag is often used for main content
    article = soup.find("article")
    if article:
        return self._clean_text(article.get_text(separator=" ", strip=True))

    # Main tag is another common container
    main = soup.find("main")
    if main:
        return self._clean_text(main.get_text(separator=" ", strip=True))

    # Fall back to looking for content by density/positioning
    # (This would contain more complex heuristics in a real implementation)

    # Basic fallback
    body = soup.find("body")
    if body:
        return self._clean_text(body.get_text(separator=" ", strip=True))

    return ""

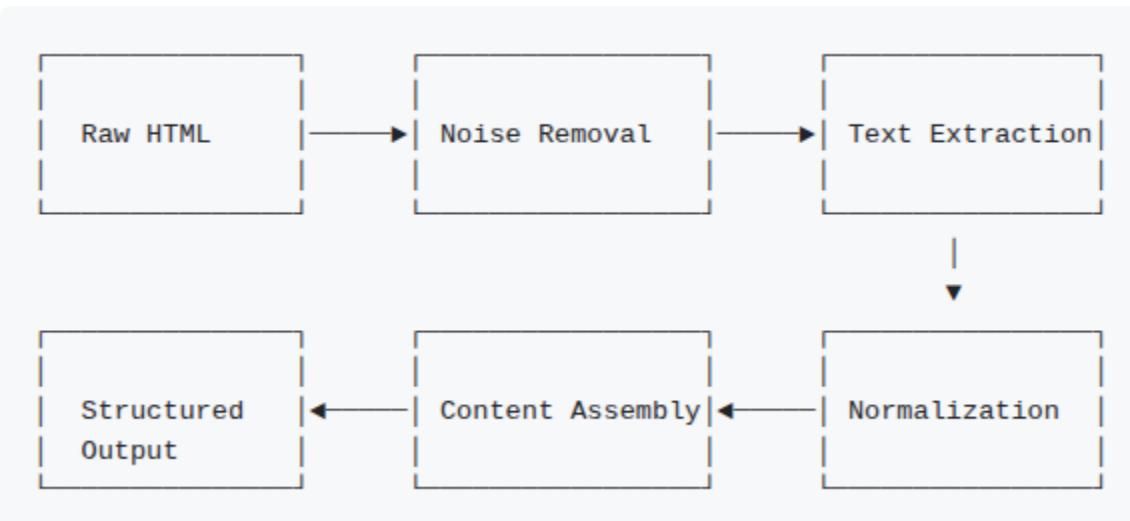
def _clean_text(self, text: str) -> str:
    """Clean and normalize text"""
    # Replace multiple whitespaces
```

```
text = " ".join(text.split())
# Additional cleaning logic...
return text
```

## Common Pitfalls

- Overly Aggressive Cleaning: Removing useful content or structural elements that are needed for context
- Static Selector Patterns: Using fixed CSS selectors that don't adapt to different website layouts
- Missing Error Fallbacks: Not providing fallback extraction methods when primary methods fail
- Inefficient DOM Traversal: Using computationally expensive selection methods for large HTML documents
- Ignoring Site-Specific Logic: Applying generic extraction logic to sites that require custom handling

## Architecture Diagram



## Further Resources

- [Mozilla Guide to DOM Manipulation](#)
- [Content Extraction Algorithms](#)
- [Text Processing with NLTK](#)
- [HTML Sanitization Best Practices](#)
- [Readability Algorithms](#) for content extraction

## 3. Multi-Layer Architecture with API Separation

### Concept Overview

Multi-layer architecture with API separation divides an application into distinct frontend and backend applications that communicate through well-defined API interfaces, allowing for independent development and deployment.

### Problem It Solves

Monolithic applications become difficult to maintain, scale, and deploy as they grow in complexity, while tightly coupled frontend and backend code limits team efficiency and flexibility.

### Solution Approach

- API Contract: Defining clear endpoints with standardized request/response formats
- Backend Independence: Building a self-contained backend application with well-defined responsibilities
- Frontend Isolation: Creating a frontend application that communicates exclusively through the API
- Cross-Origin Configuration: Implementing CORS to manage cross-domain communication

### Implementation Insight

Python

```
# Backend (FastAPI)
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, AnyHttpUrl
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

# Configure CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:8501"], # Streamlit default port
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

class AnalyzeRequest(BaseModel):
    url: AnyHttpUrl

class AnalysisResponse(BaseModel):
    title: str
    summary: str
    key_topics: list[str]
    sentiment: str
    readability_score: float

@app.post("/analyze", response_model=AnalysisResponse)
async def analyze_content(request: AnalyzeRequest):
    # Analysis logic
    return analysis_result

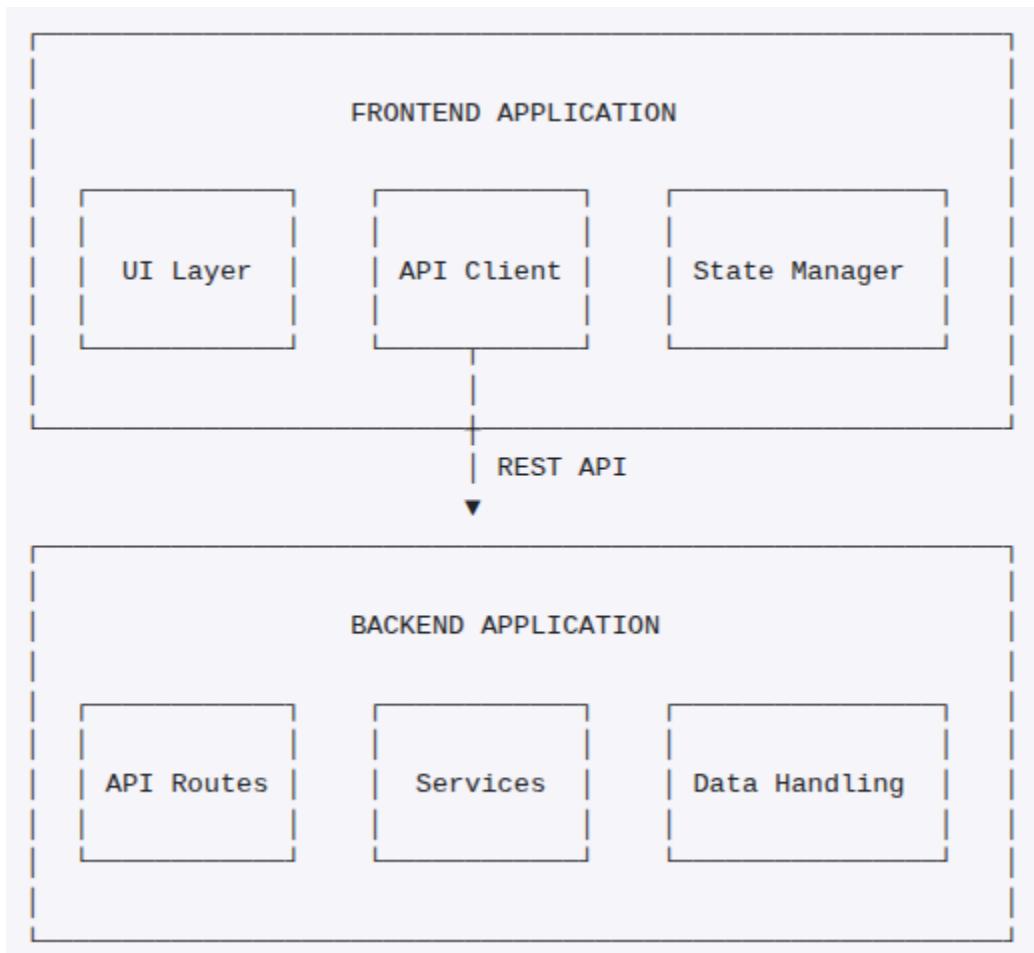
# Frontend (Streamlit) - API Client
import requests
import streamlit as st
```

```
class APIClient:  
    def __init__(self, base_url="http://localhost:8000"):  
        self.base_url = base_url  
  
    def analyze_url(self, url: str) -> dict:  
        """Send URL to backend API for analysis"""  
        try:  
            response = requests.post(  
                f"{self.base_url}/analyze",  
                json={"url": url},  
                timeout=30  
            )  
            response.raise_for_status()  
            return response.json()  
        except requests.RequestException as e:  
            raise Exception(f"API request failed: {str(e)}")  
  
    # Usage in Streamlit app  
def main():  
    st.title("Web Content Analyzer")  
    url = st.text_input("Enter a URL to analyze")  
  
    if st.button("Analyze"):  
        with st.spinner("Analyzing content..."):  
            try:  
                api_client = APIClient()  
                result = api_client.analyze_url(url)  
  
                # Display results  
                st.subheader(result["title"])  
                st.write(f"Summary: {result['summary']}")  
                # Additional display logic...  
  
            except Exception as e:  
                st.error(f"Analysis failed: {str(e)}")
```

## Common Pitfalls

- Tight Coupling: Creating dependencies between frontend and backend that reduce flexibility and maintainability
- Inconsistent API Contracts: Changing API response structures without proper versioning
- Inadequate CORS Configuration: Overly restrictive or insecure cross-origin settings
- Missing API Documentation: Poor or non-existent documentation of endpoints and expected responses
- Frontend-Backend Drift: Frontend and backend evolving independently without maintaining compatibility
- Insufficient Error Handling: Not properly propagating and handling errors between layers

## Architecture Diagram



## Further Resources

- [FastAPI Best Practices](#)
- [Streamlit-FastAPI Integration Guide](#)
- [API Design Guidelines](#)
- [CORS Documentation](#)
- [Full-Stack Patterns](#)

## 4. Security in Web Scraping (SSRF Prevention)

### Concept Overview

Server-Side Request Forgery (SSRF) prevention involves implementing safeguards to ensure that web scraping functionality cannot be exploited to access internal networks or unauthorized resources.

## Problem It Solves

Web scraping applications that blindly follow user-provided URLs can be manipulated to access internal services, localhost resources, or restricted networks, potentially leading to severe security breaches.

## Solution Approach

- URL Validation: Ensuring URLs are well-formed and use appropriate protocols
- Domain Whitelisting/Blacklisting: Restricting which domains can be scraped
- Private IP Blocking: Preventing access to internal networks and localhost
- Request Limitation: Implementing timeouts and size restrictions

## Implementation Insight

```
Python
import ipaddress
from urllib.parse import urlparse
from typing import List, Union, Optional

class URLValidator:
    def __init__(self):
        # Common private IP patterns
        self.private_ip_blocks = [
            "10.0.0.0/8",
            "172.16.0.0/12",
            "192.168.0.0/16",
            "127.0.0.0/8", # localhost
            "169.254.0.0/16", # link-local
        ]
```

```
        "::1/128", # IPv6 localhost
        "fc00::/7" # IPv6 private
    ]
    self.private_networks = [ipaddress.ip_network(block) for block in
self.private_ip_blocks]

def is_safe_url(self, url: str) -> bool:
    """Check if a URL is safe to scrape"""
    try:
        # Check URL format
        parsed = urlparse(url)
        if parsed.scheme not in ["http", "https"]:
            return False

        # Resolve domain to IP
        domain = parsed.netloc.split(":")[0] # Remove port if present

        try:
            # Convert domain to IP address
            ip = ipaddress.ip_address(domain)

            # Check if IP is in private ranges
            for network in self.private_networks:
                if ip in network:
                    return False
            except ValueError:
                # Domain is not an IP address, proceed with other checks
                pass

        # Additional checks can be added here
        return True

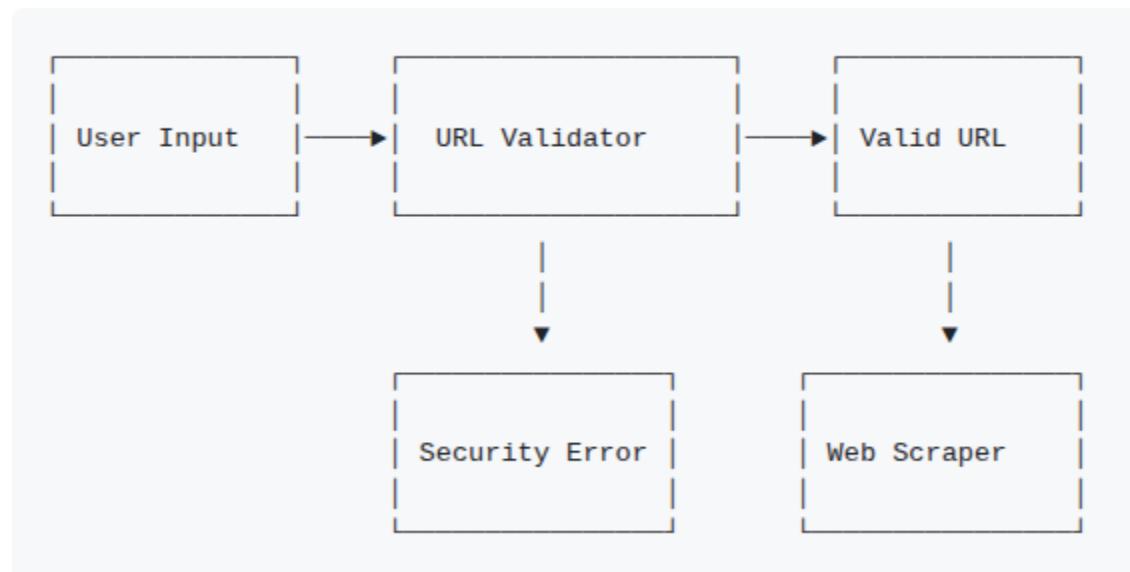
    except Exception as e:
        # If any error occurs, consider the URL unsafe
        logger.error(f"URL validation error: {str(e)}")
```

```
return False
```

## Common Pitfalls

- Incomplete IP Validation: Missing certain private IP ranges or IPv6 addresses
- DNS Rebinding Vulnerability: Not checking for DNS rebinding attacks that initially resolve to safe IPs
- Insufficient Scheme Validation: Only checking http/https but missing other potentially dangerous schemes
- Bypass Through Redirects: Not following redirects to check final destination safety
- Open Redirect Exploitation: Not validating URLs that might redirect to internal networks
- Lack of Timeout Controls: No limits on request duration, potentially enabling resource exhaustion attacks

## Architecture Diagram



## Further Resources

- [OWASP SSRF Prevention Cheat Sheet](#)
- [Port Scanning with SSRF](#)
- [DNS Rebinding Attacks](#)
- [URL Validation Best Practices](#)
- [Comprehensive SSRF Guide](#)

## 5. Structured Content Analysis with LLMs

### Concept Overview

Structured content analysis with LLMs involves using large language models to analyze web content and produce structured, actionable insights rather than simple text responses.

### Problem It Solves

Raw web content is vast and unstructured, making it difficult to extract meaningful insights, summaries, or specific types of information without significant manual analysis.

## Solution Approach

- Prompt Engineering: Designing clear, specific prompts for LLMs to produce structured outputs
- Output Formatting: Requesting specific formats (JSON, structured lists) from LLMs
- Multi-dimensional Analysis: Analyzing content across various dimensions (topics, sentiment, readability)
- Context Management: Breaking large content into manageable chunks while maintaining context

## Implementation Insight

Python

```
class ContentAnalyzer:  
    def __init__(self, llm_service):  
        self.llm_service = llm_service  
  
    @asyncio.coroutine  
    def analyze_content(self, content: dict) -> dict:  
        """Analyze content using LLM"""  
        # Prepare the content for analysis  
        processed_text = self._prepare_content_for_analysis(content)  
  
        # Create structured analysis prompt  
        prompt = f"""  
        Analyze the following web content and provide a structured analysis:  
  
        TITLE: {content['title']}  
  
        CONTENT:  
        {processed_text}  
  
        Provide your analysis in the following format:  
        1. Brief summary (2-3 sentences)  
        2. Main topics (list of 3-5 key topics)  
        3. Content sentiment (positive, neutral, negative)  
        """
```

4. Readability assessment (easy, moderate, complex)
5. Key takeaways (3-5 bullet points)

Format your response as JSON with the following structure:

```
{}  
  "summary": "...",  
  "topics": ["topic1", "topic2", ...],  
  "sentiment": "...",  
  "readability": "...",  
  "takeaways": ["point1", "point2", ...]  
}  
"""
```

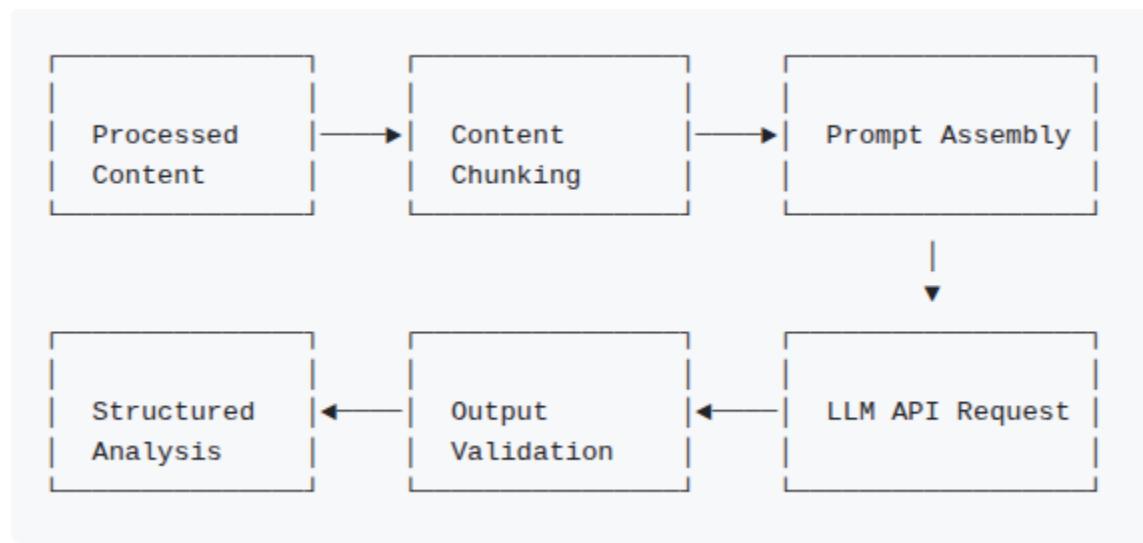
```
# Request analysis from LLM  
try:  
    result = await self.llm_service.generate_structured_response(prompt)  
    return result  
except Exception as e:  
    logger.error(f"Analysis error: {str(e)}")  
    raise ContentAnalysisException("Failed to analyze content")  
  
def _prepare_content_for_analysis(self, content: dict) -> str:  
    """Prepare content for LLM analysis"""  
    # Truncate to fit token limits  
    max_length = 5000 # Adjust based on model limits  
  
    # Combine most important elements  
    text = f"{content['title']}\n\n"  
  
    # Add headings for structure  
    for heading in content.get('headings', [])[-5:]:  
        text += f"{heading}\n"  
  
    # Add main text (truncated if necessary)  
    main_text = content.get('main_text', '')
```

```
if len(main_text) > max_length:  
    text += f"{main_text[:max_length]}..."  
else:  
    text += main_text  
  
return text
```

## Common Pitfalls

- Prompt Inconsistency: Using unclear or inconsistent prompts that lead to unpredictable outputs
- Insufficient Chunking: Not properly dividing large content into manageable chunks for token limits
- Missing Output Validation: Not validating that LLM outputs match the expected structure
- Context Loss: Losing important context when processing long documents
- Poor Error Recovery: Not having fallbacks when LLM responses are invalid or incomplete
- Token Inefficiency: Wasting tokens on irrelevant content that doesn't contribute to analysis quality

## Architecture Diagram



## Further Resources

- [OpenAI API Best Practices](#)
- [JSON Mode Guide](#)
- [Content Chunking Strategies](#)
- [Prompt Engineering Guide](#)
- [LLM Response Validation Techniques](#)

## 6. Advanced Streamlit UI Components

### Concept Overview

Advanced Streamlit UI components involve creating more sophisticated user interfaces that go beyond basic inputs and outputs, incorporating interactive elements, proper state management, and responsive layouts.

### Problem It Solves

Basic UI elements often provide poor user experience for complex applications, lacking proper organization, feedback mechanisms, and visual hierarchy needed for intuitive data exploration.

## Solution Approach

- Layout Management: Using columns, expanders, and containers for organized content display
- Progress Indicators: Implementing spinners and progress bars for long-running operations
- Interactive Elements: Adding tabs, selectors, and interactive visualizations
- Conditional Rendering: Showing different UI elements based on application state

## Implementation Insight

Python

```
import streamlit as st
import pandas as pd
import altair as alt

def display_analysis_report(analysis_result: dict):
    """Display a comprehensive analysis report with advanced UI
    components"""

    # Header section
    st.header(analysis_result["title"])
    st.write(f"**URL**: {analysis_result['url']}")

    # Summary in an expander
    with st.expander("Summary", expanded=True):
        st.write(analysis_result["summary"])

    # Use tabs for different analysis sections
    tab1, tab2, tab3 = st.tabs(["Content Analysis", "SEO Insights", "Readability"])

    with tab1:
        # Use columns for layout
        col1, col2 = st.columns([2, 1])
```

```
with col1:  
    st.subheader("Main Topics")  
    for topic in analysis_result["topics"]:  
        st.write(f"• {topic}")  
  
with col2:  
    # Create a sentiment gauge  
    sentiment = analysis_result["sentiment"]  
    sentiment_value = {"positive": 0.8, "neutral": 0.5, "negative":  
0.2}.get(sentiment, 0.5)  
  
    st.subheader("Content Sentiment")  
    st.progress(sentiment_value)  
    st.write(f"**{sentiment.capitalize()}**")  
  
with tab2:  
    # SEO metrics visualization  
    st.subheader("SEO Analysis")  
  
    # Create sample data for visualization  
    seo_data = pd.DataFrame({  
        'Metric': ['Title Length', 'Keyword Density', 'Link Structure', 'Meta  
Quality'],  
        'Score': [  
            analysis_result.get("seo_scores", {}).get("title_score", 75),  
            analysis_result.get("seo_scores", {}).get("keyword_score", 60),  
            analysis_result.get("seo_scores", {}).get("link_score", 80),  
            analysis_result.get("seo_scores", {}).get("meta_score", 65)  
        ]  
    })  
  
    # Create bar chart  
    chart = alt.Chart(seo_data).mark_bar().encode(  
        x=alt.X('Score:Q', scale=alt.Scale(domain=[0, 100])),  
        y=alt.Y('Metric:N', sort='-x'),
```

```
color=alt.condition(
    alt.datum.Score > 70,
    alt.value('green'),
    alt.value('orange')
)
).properties(height=200)

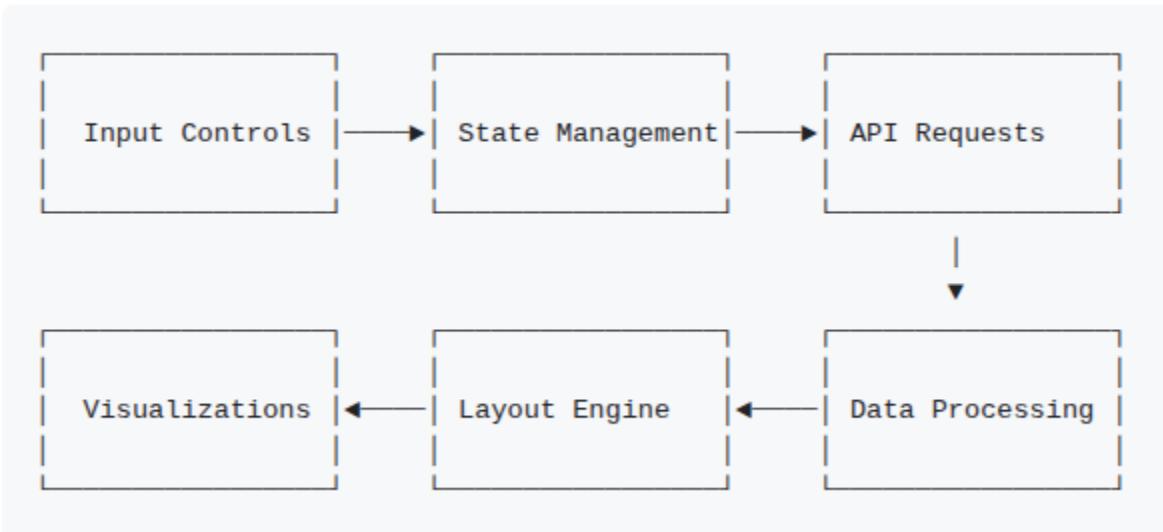
st.altair_chart(chart, use_container_width=True)

with tab3:
    # Readability metrics
    st.subheader("Readability Analysis")
    st.write(f"**Level**: {analysis_result.get('readability', 'Moderate')}\n\n"
            "# Show key takeaways"
            st.subheader("Key Takeaways")
            for i, point in enumerate(analysis_result.get("takeaways", [])):
                st.write(f"{i+1}. {point}")
```

## Common Pitfalls

- State Management Issues: Not properly handling Streamlit's session state, leading to data loss on page refresh
- Performance Bottlenecks: Running expensive operations directly in the UI code instead of using caching
- Overcomplicated Layouts: Creating confusing or cluttered interfaces with too many components
- Missing Loading States: Not showing appropriate loading indicators during processing
- Poor Error Feedback: Displaying technical error messages instead of user-friendly guidance
- Responsive Design Issues: Not testing and optimizing for various screen sizes and device types

## Architecture Diagram



## Further Resources

- [Streamlit Documentation](#)
- [Streamlit Session State Guide](#)
- [Streamlit Caching](#)
- [Streamlit Layouts](#)
- [Streamlit Components Library](#)
- [Data Visualization with Streamlit](#)