Functionally, POML components fall into several key categories, conceptually similar to how complex user interface widgets are built upon fundamental HTML elements.

- **Basic Structural Components** provide fundamental text formatting and structural grouping capabilities, analogous to common HTML tags. Elements like `<b>` (bold), `<i>` (italic), `<p>` (paragraph), and `<div>` (division) allow for control over text presentation and logical grouping within larger content blocks, enhancing readability and structure. Alongside these static elements, POML includes components and syntax for dynamic content generation through its integrated templating engine, such as constructs for loops (`for`), conditionals (`if`), variable definition (`<let>`), and substitution (`{{...}}`). These templating features are detailed further in § 4.4.
- **Intention Components** define the core logic and overall structure of the prompt. These components implement the "structured prompting" paradigm [30, 51, 68, 95, 104], establishing the interaction's purpose and guiding the LLM's response. Examples include the `<role>` component, used to define the persona the LLM should adopt; the `<task>` component, which specifies the objective the LLM needs to achieve; and the `<example>` component, crucial for providing few-shot demonstrations to guide the model's behavior. Unlike data components, intention components organize the logical blocks of the prompt rather than presenting complex data directly. When rendered, components like `<role>` typically appear as formatted titles (e.g., `**Role:**`), with the presentation adjustable via styling rules (§ 4.3). These intention components orchestrate the overall flow and content of the prompt, as illustrated in Figure 2.
- **Data Components** are designed to handle the integration of diverse external data formats into the prompt context. Elements such as `<document>`, `<img>`, and `<table>` provide methods to embed content from files or data structures. These components, detailed in § 4.2, are essential for grounding LLM responses in specific information or enabling tasks that operate on external data.

Rationale This hierarchical and descriptive markup enforces a logical organization, offering significant advantages over unstructured plain text. For instance, as illustrated in Figure 2, the explicit separation of conceptual parts — using intention components like `<role>` and `<task>`, structuring instructions with `<stepwise-instructions>`, defining outputs with `<output-format>`, and providing few-shot demonstrations [14] via `<example>` (which itself nests `<input>` containing data components like `<table>` and `<output>` referencing an external `<doc>`) — significantly improves prompt clarity and understandability, reducing ambiguity for both LLMs and human readers. These components, such as the tables shown integrated directly or within examples, can then be easily reused across different prompts or modified in isolation, fostering systematic prompt engineering (**DG1**). Compared to simpler formats like PromptML [68], which primarily offer basic task and example definitions, POML provides richer semantic components crucial for handling diverse content types (**DG2**). Meanwhile, the modularity allows individual components to be treated as self-contained units. Modifying components independently minimizes the risk of unintended side effects elsewhere in the prompt, facilitating faster and safer iterations, supporting more agile development workflows. The combination of a standardized, text-based format and clearly named elements simplifies prompt management within standard version control systems like Git (Figure 1 (a)). This enhances collaboration, as changes become easier to track, merge, and discuss (**DG4**).

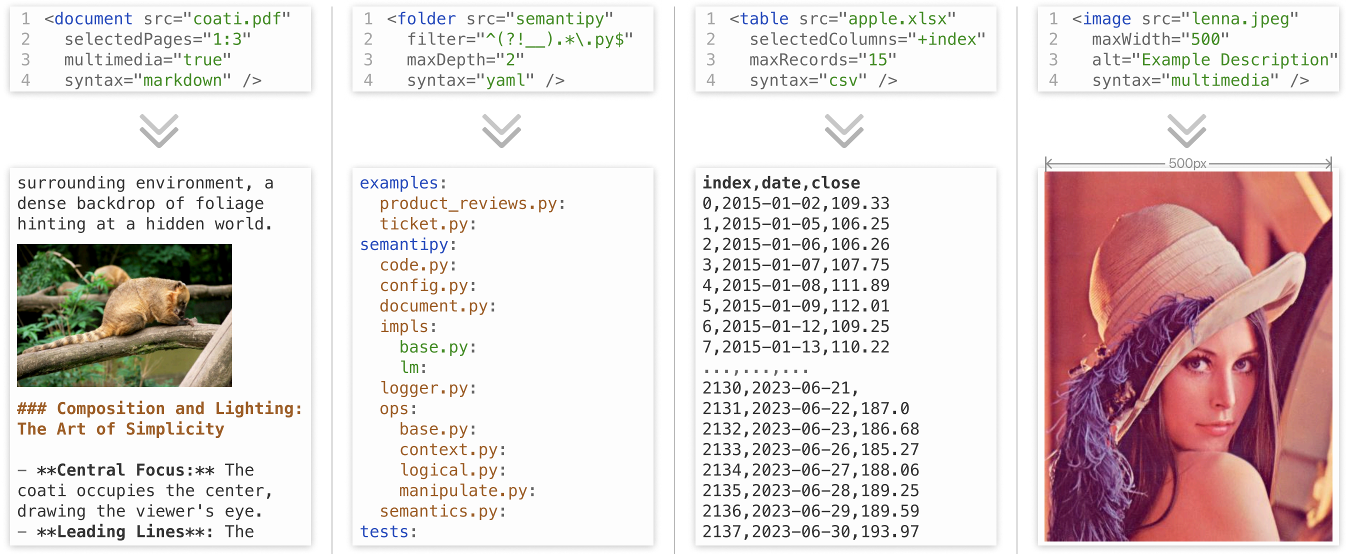
4.2 Data Components

POML provides specialized data components to systematically integrate diverse data modalities directly within the prompt structure, as exemplified in Figure 3. This capability directly addresses the critical need to combine LLM reasoning with varied external data sources, a key requirement for many advanced applications (**DG2**). POML's modular data components reduce the inherent complexity associated with constructing prompts that incorporate multiple data inputs. These built-in components distinguish POML by offering systematic handling for common data types within a unified framework, analogous to component libraries in modern UI frameworks [23, 58]. Currently, POML supports **7 distinct data components**: document, table, folder, image, conversational messages, audio clip, web page. We detail **4 representative examples** here; the remaining components operate similarly and are demonstrated in the case studies (§ 7).

Document The document component (Figure 3 (a)) provides a mechanism for referencing and embedding content from external text-based files, supporting formats like `.txt`, `.docx`, and `.pdf`. This is particularly crucial for tasks involving large amounts of background information or context, such as in retrieval-augmented generation (RAG) systems [16, 45]. It offers granular control through attributes, allowing developers to specify character encoding, whether to preserve original formatting, whether to include or discard embedded images within documents like PDFs, and how to trim content — for instance, by selecting specific page ranges (e.g., `selectedPages="1:3"`) — thereby helping manage the LLM's limited context window.

Folder For scenarios involving interaction with file systems or code repositories, such as file system analysis, code reviews, and project navigation [39, 47, 86], the `<folder>` component (Figure 3 (b)) provides a structured way to represent directory hierarchies. It allows customization of the representation through attributes controlling the maximum depth of the displayed tree, filtering options to include or exclude files based on extensions or name patterns (e.g., `filter="^(?!__).*\.py"`), and optional automatic summarization for very large directories to conserve context space. It can also display metadata like file sizes or modification dates and supports multiple output formats, including classic tree views using box-drawing characters, YAML, or JSON representations.

Table Tables are vital for complex question-answering over structured data [64], generating data visualizations [18], or performing data manipulation tasks [103]. The `<table>` component (Figure 3 (c)) supports a variety of input sources, including CSV, TSV, Excel spreadsheets, and JSON arrays of objects. Crucially, it allows specifying the output representation (e.g., Markdown, HTML, XML, plain CSV) and controlling content details such as the inclusion of



```
<document src="coati.pdf"
  selectedPages="1:3"
  multimedia="true"
  syntax="markdown" />
```

```
surrounding environment, a
dense backdrop of foliage
hinting at a hidden world.
```

```
### Composition and Lighting:
The Art of Simplicity

- **Central Focus:** The
coati occupies the center,
drawing the viewer's eye.
- **Leading Lines**: The
```

(a) Document Component Usage

```
<folder src="semantipy"
  filter="^(?!__).*\.py$"
  maxDepth="2"
  syntax="yaml" />
```

```
examples:
  product_reviews.py:
  ticket.py:
semantipy:
  code.py:
  config.py:
  document.py:
  impls:
    base.py:
    lm:
  logger.py:
  ops:
    base.py:
    context.py:
    logical.py:
    manipulate.py:
  semantics.py:
tests:
```

(b) Folder Component Usage

```
<table src="apple.xlsx"
  selectedColumns="+index"
  maxRecords="15"
  syntax="csv" />
```

```
index,date,close
0,2015-01-02,109.33
1,2015-01-05,106.25
2,2015-01-06,106.26
3,2015-01-07,107.75
4,2015-01-08,111.89
5,2015-01-09,112.01
6,2015-01-12,109.25
7,2015-01-13,110.22
...,...,...
2130,2023-06-21,
2131,2023-06-22,187.0
2132,2023-06-23,186.68
2133,2023-06-26,185.27
2134,2023-06-27,188.06
2135,2023-06-28,189.25
2136,2023-06-29,189.59
2137,2023-06-30,193.97
```

(c) Table Component Usage

```
<image src="lenna.jpeg"
  maxWidth="500"
  alt="Example Description"
  syntax="multimedia" />
```

(d) Image Component Usage

Figure 3: Examples of POML data components demonstrating integration of diverse data types (§ 4.2). (a) `<document>` rendering selected PDF pages with multimedia; (b) `<folder>` displaying a filtered directory structure as YAML; (c) `<table>` extracting and formatting spreadsheet data as CSV; (d) `<img>` inserting a referenced image with resizing.

headers or indices, and selective presentation of rows or columns for large tables, offering flexibility in presentation. Employing a consistent and well-defined table format via this component can significantly enhance an LLM's ability to accurately parse and reason over the tabular data, applicable to scenarios like [36, 80, 103].

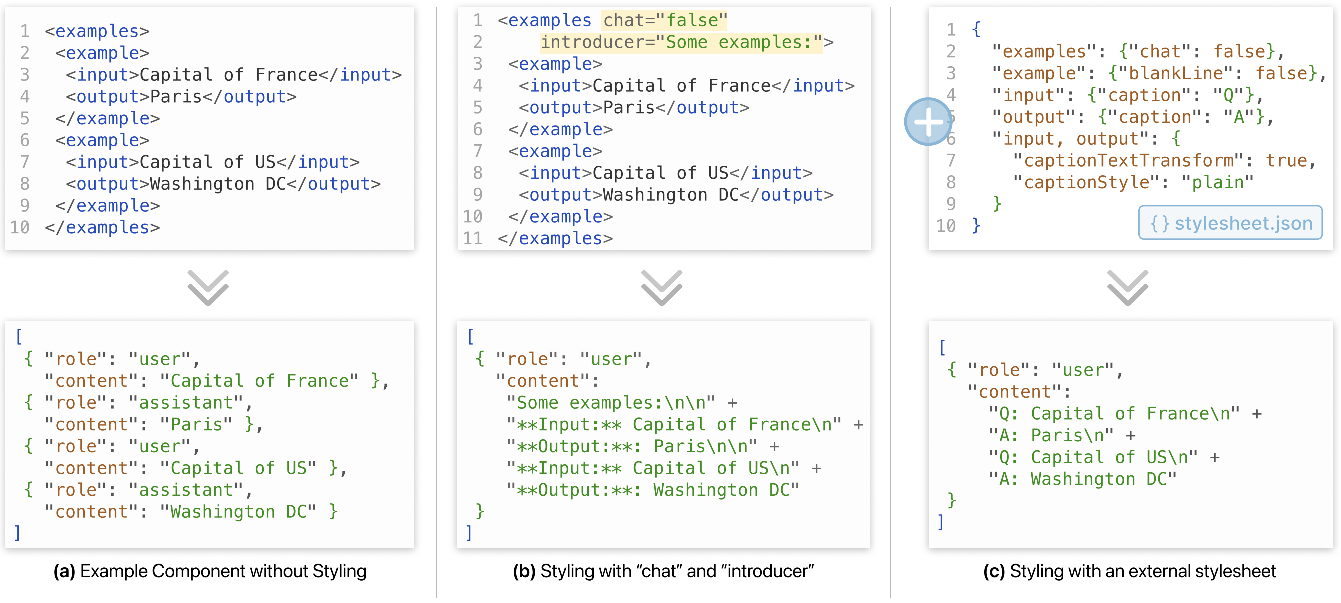
Image Acknowledging the rapid advancement of multimodal LLMs capable of processing visual information [62, 83], POML includes the `<img>` component (Figure 3 (d)) for direct image insertion into prompts. Inspired by web accessibility principles [92–94], this component supports standard attributes like `alt` text, allowing prompts to be easily adapted to LLMs with varying capabilities, including those without vision, thereby enhancing robustness. Layout can be influenced using a `position` attribute (e.g., `before`, `after`, `here`) to control placement relative to surrounding text. Attributes like `maxWidth` and `maxHeight` allow developers to suggest rendering constraints, potentially influencing the number of tokens consumed when sending image data to the model. Compared to interactive uploads in chat interfaces [21, 60] or less structured parameter passing in raw API calls [62, 83], inserting images via a dedicated component within the markup offers superior programmatic control and integration.

Extensions POML's architecture is designed to be extensible. As LLMs evolve to handle new modalities, POML can incorporate additional data components for types like video segments and node-edge graphs, maintaining a consistent framework for diverse data integration.

4.3 Styling System

A critical challenge in prompt engineering stems from the documented sensitivity of LLMs to subtle variations in input formatting [35, 73, 76, 89, 106]. Minor changes in presentation can significantly impact model performance, necessitating precise control over how prompt content is rendered (**DG3**). POML addresses this through a dedicated styling system designed to manage presentation effectively. The styling options provided, such as control over syntax formats, layout adjustments (e.g., chat vs. block formatting), list styles, and caption presentation, are informed by findings in prompt engineering research exploring these sensitivities [48, 76]. Inspired by the separation of concerns in web development (HTML for structure, CSS for style [26, 94]), POML deliberately decouples presentation rules from the core prompt content. POML offers two primary mechanisms for applying styles: *inline attributes* and *external stylesheets*.

POML provides control over various presentation aspects known to influence LLM behavior, such as layout adjustments (e.g., chat versus block formatting), list styles (`<list listStyle="decimal">`), caption presentation (`<hint caption="Important Note" captionStyle="header">`), and overall verbosity [48, 76]. Beyond simple formatting, POML provides a crucial **`syntax` attribute**. This attribute allows developers to explicitly control the rendering format for specific components or the entire prompt (e.g., `<table syntax="html"/>` or `<poml syntax="json">`). This control is vital because different LLMs can exhibit varying sensitivities or capabilities when parsing structured formats like Markdown, JSON, or XML [7] embedded within the prompt text. The styling attributes can be nested, allowing, for example, a subsection of a predominantly Markdown-formatted prompt to be rendered as JSON, providing fine-grained control over the final string sent to the model. One way to apply these styling adjustments is via **inline attributes** on specific POML components — a convenient method for localized formatting familiar to web developers. Figure 4 ((b) compared to



```
<examples>
 <example>
  <input>Capital of France</input>
  <output>Paris</output>
 </example>
 <example>
  <input>Capital of US</input>
  <output>Washington DC</output>
 </example>
</examples>
```

```
[
 { "role": "user",
   "content": "Capital of France" },
 { "role": "assistant",
   "content": "Paris" },
 { "role": "user",
   "content": "Capital of US" },
 { "role": "assistant",
   "content": "Washington DC" }
]
```

(a) Example Component without Styling

```
<examples chat="false"
    introducer="Some examples:">
 <example>
  <input>Capital of France</input>
  <output>Paris</output>
 </example>
 <example>
  <input>Capital of US</input>
  <output>Washington DC</output>
 </example>
</examples>
```

```
[
 { "role": "user",
   "content":
    "Some examples:\n\n" +
    "**Input:** Capital of France\n" +
    "**Output:**: Paris\n\n" +
    "**Input:** Capital of US\n" +
    "**Output:**: Washington DC"
 }
]
```

(b) Styling with "chat" and "introducer"

```
{
  "examples": {"chat": false},
  "example": {"blankLine": false},
  "input": {"caption": "Q"},
  "output": {"caption": "A"},
  "input, output": {
    "captionTextTransform": true,
    "captionStyle": "plain"
  }
}
```

```
[
 { "role": "user",
   "content":
     "Q: Capital of France\n" +
     "A: Paris\n" +
     "Q: Capital of US\n" +
     "A: Washington DC"
 }
]
```

(c) Styling with an external stylesheet

Figure 4: Demonstrating POML styling capabilities (§ 4.3). (a) Default rendering of `<example>` components. (b) Inline `chat` and `introducer` attributes on the parent `<examples>` element modify its presentation (from chat messages to plain text with "`**Input**`"/"`**Output**`" captions). (c) A `<stylesheet>` applies global rules to POML in (a), controlling layout (`chat=false`), captions/prefixes (`caption="Q:"`/`"A:"`), and styles (`captionStyle="plain"`), resulting in customized output.

(a)) demonstrates how inline attributes can alter the presentation of `<example>` components.

For managing styles more systematically across multiple components or prompts, POML supports **external stylesheets**, typically defined in separate JSON files (e.g., `stylesheet.json` as shown in Figure 4 (c)). These files contain global formatting rules defined using a concise, JSON-like syntax. Style rules within the stylesheet target specific POML component types (e.g., `hint`, `table`) or user-defined classes (see § 7.2 for examples). Alternatively, these style rules can also be embedded directly within a POML document using the `<stylesheet>` tag, as shown in Figure 1 (c). Overall, stylesheets provide a mechanism to define styles that apply globally or to specific component types, offering a way to batch-manage or customize styles that might otherwise be set inline individually. They offer several advantages: (1) it provides centralized control, establishing a single source of truth for formatting; (2) it keeps the primary prompt logic cleaner by avoiding repetitive presentation attributes; (3) it simplifies style modifications by eliminating multi-point edits, enhancing maintainability. As a result, this approach directly facilitates systematic experimentation with different presentation formats to address LLM sensitivities or task requirements (**DG3**) without altering the core prompt content.

4.4 Templating Engine

POML integrates a built-in templating engine to facilitate the creation of dynamic, data-driven prompts without execution environment dependencies (**DG2**). Inspired by established web templating systems [29, 34, 40, 69], this engine allows runtime customization without relying solely on external scripting. Key features include

```
<poml>
  <task>Review a collection of text files.
    Generate a summary of their key themes.</task>
  <hint>Large files (&gt;10KB) can be skipped.</hint>
  <let name="files" src="files.json" />
  <div for="file in files">
    <p>File name: {{ file.name }}</p>
    <document if="file.size < 10*1024" src="{{ file.path }}" />
    <p else>File size is too large ({{ file.size / 1024 }} KB).</p>
  </div>
</poml>
```

Figure 5: Example of POML's templating engine (§ 4.4): using `<let>` to load data (from `files.json`), `for` attribute to iterate over items, `{{ ... }}` for variable substitution (e.g., `{{ file.name }}`), and `if`/`else` attributes for conditional component rendering based on data values (embedding a `<document>` only if `file.size` is below a threshold)

variable substitution using `{{variable}}` syntax, iteration over data collections via the `for` attribute, conditional rendering using `if`/`else`-like attributes, and inline variable definition with the `<let>` tag. Figure 5 demonstrates these capabilities, showing how data loaded via `<let>` is iterated over with `for`, variables like `{{ file.name }}` are substituted, and component rendering is controlled conditionally based on `file.size`.

This integrated approach offers advantages over manual string manipulation, which is often error-prone and hard to debug [15]. It reduces redundancy by enabling reusable prompt structures populated with varying data. The declarative nature enhances readability, providing a clearer view of the prompt structure. Furthermore, the engine is independent from other programming languages, differentiating POML from other solutions relying on existing templating