Code Style Guide

Product Name: LEMMA
Team Name: LEMMA
Revision Date: May 24, 2025

This guide describes the coding conventions that are expected during the development of LEMMA. The primary languages utilized in this product are TypeScript and JavaScript.

Naming

Variables and functions

All variable and function names are written in camel case. For example:

```
let isStreaming: boolean = false;
export const startStreaming = (): void => {isStreaming = true};
```

Classes and methods

Class names are written in upper camel case, while method names are written in lower camel case. For example:

```
export class InferenceService {...
  async getChunks = (content: string, filename: string) => {
    ...
  };
}
```

Interfaces and Types

All interface and type names are written in upper camel case. For example:

```
export type FileType = 'user' | 'assisted' | 'generated';
interface Note {...}
```

Source files

The naming convention for source files within the `main` directory (backend) is in kebab case, while for source files within the `renderer` directory (frontend) is in camel case.

Formatting

Indentation

All source files follow a 2-space indentation in every line.

Functions

All functions are formatted in arrow function expressions.
Example:

```
const setupIpcHandlers = (): void => {...};
```

Strings

All strings are encapsulated with single quotes, except where string interpolation is needed—in which case, backticks are used.

Line length/wrapping

Any lines of code that appear too long (more than 130 characters) should be wrapped on subsequent lines with an extra indentation of 2 spaces.

Objects

Objects may be defined inline if the properties are shortly defined. Otherwise, the properties should be wrapped in subsequent lines. For example:

```
const data = { content: 'hello, world!' };
const testQuery = {
  similarityQuery: 'steps to compile',
};
```

Arrays

Arrays may be defined inline if the elements are shortly defined. Otherwise, the elements should be wrapped in subsequent lines. For example:

```
const messages = [systemPrompt, userPrompt];
const messages = [
  { role: "system", content: "You are an AI assistant." },
  { role: "user", content: prompt }
];
```

Ternary operations

If the ternary operation is too long, the Then-expression and the Else-expression of the ternary operation should be wrapped in subsequent lines. For example:

```
const chromaRunCommand = process.platform === 'win32'
    ? path.join(venvPath, 'Scripts', 'chroma.exe')
    : path.join(venvPath, 'bin', 'chroma');
```

Logical expressions

Logical expressions that appear too long should be broken into subexpressions, with each wrapped in subsequent lines. For example:

```
if (!(Array.isArray(filePath) && Array.isArray(content))
  || (Array.isArray(filePath) && Array.isArray(content)) {
  throw TypeError('upsertNote(): mismatched types.');
}
```

Function/method parameters

Functions and methods with many or long parameters should have the parameters wrapped in subsequent lines. For example:

```
public upsertNotes = async (
    notesDirectory: string,
    filePath: string | string[],
    content: string | string[],
): Promise<void> => {...};
```

Comments/Document strings

Functions, Classes, and Methods

Functions, classes, and methods should be preceded by multi-line comments that describe their purpose and behavior, allowing code editors to display helpful documentation when hovering over their definition. For example:

```
/**
 * Send a chat completion request to chunk the document
 */
const getChunks = async (content: string, filename: string) {};
```

Annotations

Code blocks should be frequently annotated by single-line comments (starting with double slashes) that briefly describe the functionality of the code execution.