Automated Refactoring Pipeline Evaluation Bonus

Team 8

The analysis of the pipeline can be split into two section - Smell Detection and Code Refactoring.

Testing Conditions

Files Used

The test was conducted on files from **/reader-web/src/main/java/com/sismics/reader/rest/resource** , due to api limit restrictions the task was limited to two files **LocaleResource.java** and **AppResource.java**. These files were in the folder concerning **Issue #2** for task 3a.

LLM's Used

Gemini-1.5 and Deepseek-r1

Prompts Used

Code smells

```
prompt = (
            "Analyze the following Java code for common code smells such as "
            "long methods, duplicated blocks, or inappropriate naming.\n"
            f"Code:\n{file_content}\n"
            "Provide a concise summary in the form of bullet for each potential
code smell."
        )
```

Design Smells

```
prompt = (
            "Analyze the following Java code for design smells such as large
classes, "
            "god objects, circular dependencies, or poor abstraction.\n"
            f"Code:\n{file_content}\n"
            "Provide a concise summary in the form of bullet for each potential
design smell."
        )
```

Code Refactoring

```
        prompt = (
            "Given the following Java code along with identified code smells
and design smells, "
            "refactor the code to address these issues while preserving
functionality . give me only fully working java code as output , no
explanations.\n"
            f"Code Smells:\n{code_smells_report}\n\n"
            f"Design Smells:\n{design_smells_report}\n\n"
            "Original Code:\n"
            f"{file_content}\n\n"
            "Provide only the working refactored code in correct Java syntax."
        )
```

Metrics used for Refactoring

- Lines of Code
- Cyclomatic Complexity

Smell Detection

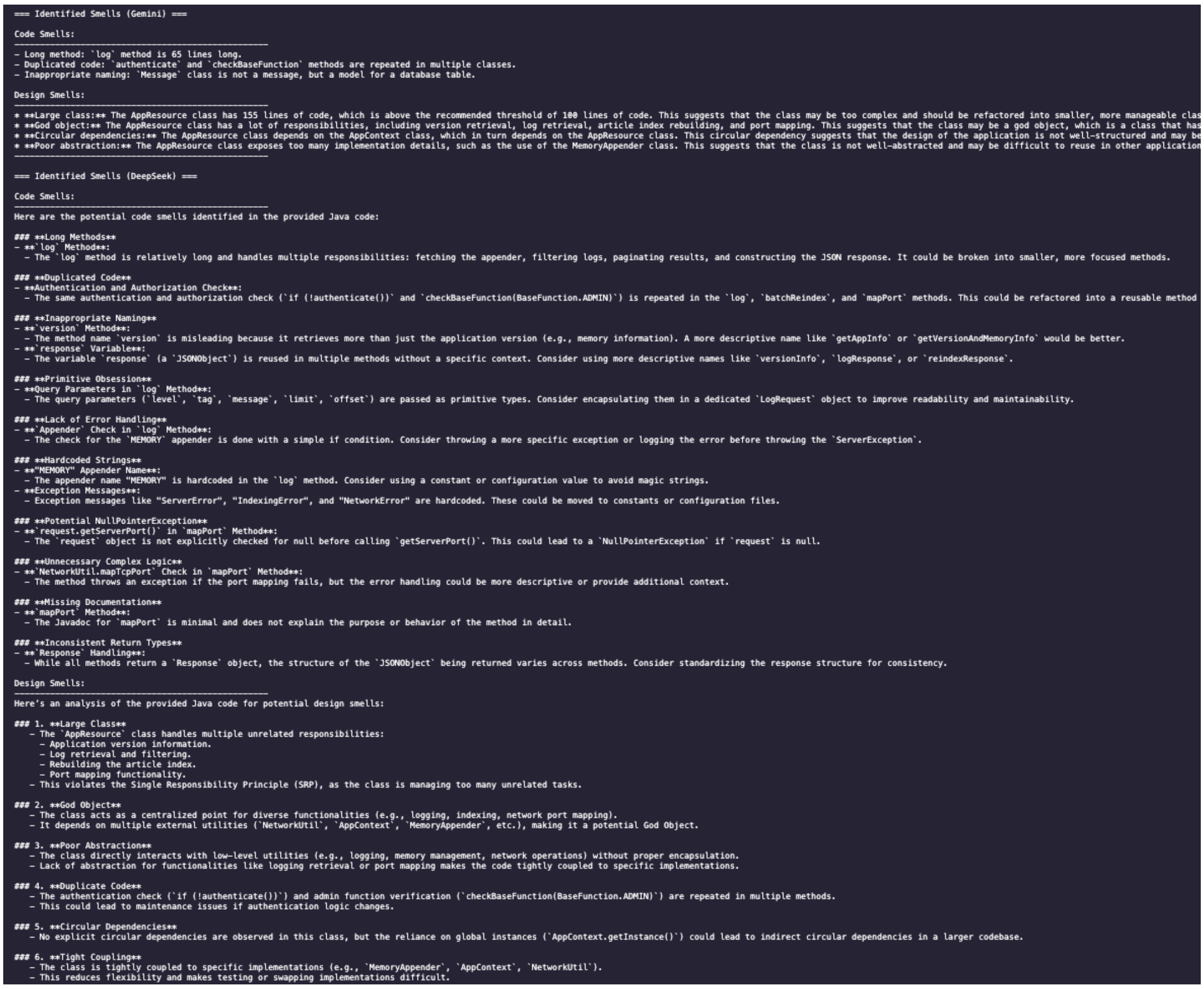
Smell Detection Results

For the smell detection task, there was a notable difference between the two models. **Gemini** demonstrated greater consistency in identifying smells across multiple runs. However, the smells it detected and their descriptions were relatively shorter and less detailed. Thismay have limited the context provided to the LLM during the code refactoring process, potentially impacting the quality of its suggestions.

On the other hand, **DeepSeek** provided significantly more detailed descriptions of the detected smells, likely due to its nature as a reasoning-based model. While the higher number of detected smells could be seen as a strength, it also raised concerns, as some of the identified smells appeared to be hallucinations. Moreover, DeepSeek exhibited inconsistency by failing to detect any smells during certain runs for the same file, indicating that it may not be a reliable choice for refactoring tasks.

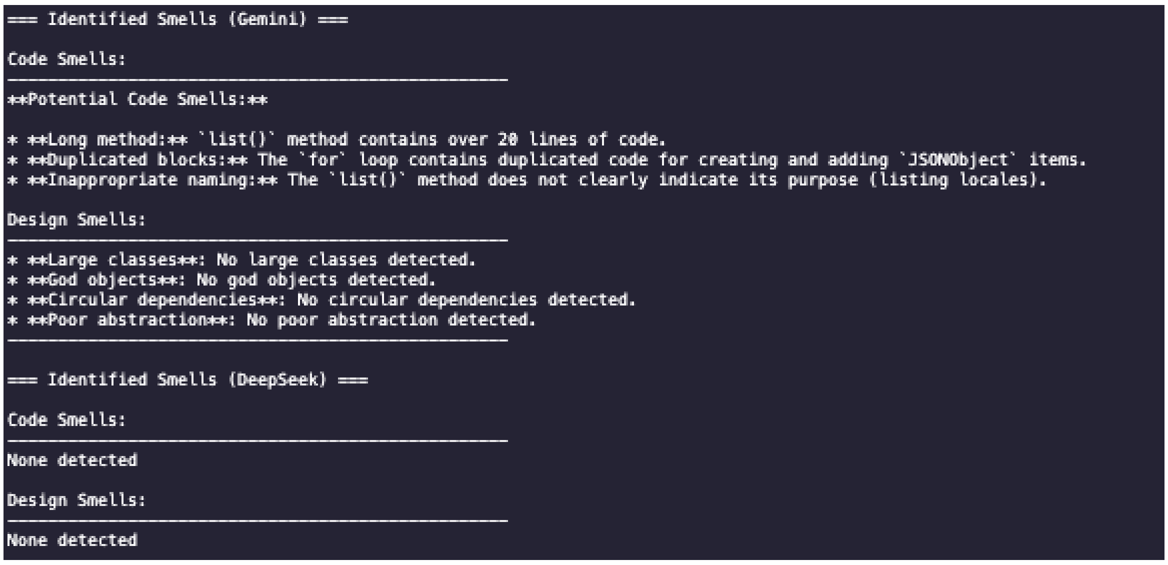
Observations

- Gemini gave fewer smells with shorter descriptions however it was more consistent in identifying smells, and suffered from fewer hallucinations.
- DeepSeek performed significantly better at identifying smells in some runs, however on average it did not perform satisfactorily. A general trend noticed was that DeepSeek sometimes identified smells that did not exist in the code



```
=== Identified Smells (Gemini) ===

Code Smells:
--------------------------------------------------
- Long method: `log` method is 65 lines long.
- Duplicated code: `authenticate` and `checkBaseFunction` methods are repeated in multiple classes.
- Inappropriate naming: `Message` class is not a message, but a model for a database table.

Design Smells:
--------------------------------------------------
* **Large class:** The AppResource class has 155 lines of code, which is above the recommended threshold of 100 lines of code. This suggests that the class may be too complex and should be refactored into smaller, more manageable clas
* **God object:** The AppResource class has a lot of responsibilities, including version retrieval, log retrieval, article index rebuilding, and port mapping. This suggests that the class may be a god object, which is a class that has
* **Circular dependencies:** The AppResource class depends on the AppContext class, which in turn depends on the AppResource class. This circular dependency suggests that the design of the application is not well-structured and may be
* **Poor abstraction:** The AppResource class exposes too many implementation details, such as the use of the MemoryAppender class. This suggests that the class is not well-abstracted and may be difficult to reuse in other application
--------------------------------------------------

=== Identified Smells (DeepSeek) ===

Code Smells:
--------------------------------------------------
Here are the potential code smells identified in the provided Java code:

### **Long Methods**
- **`log` Method**:
  - The `log` method is relatively long and handles multiple responsibilities: fetching the appender, filtering logs, paginating results, and constructing the JSON response. It could be broken into smaller, more focused methods.

### **Duplicated Code**
- **Authentication and Authorization Check**:
  - The same authentication and authorization check (`if (!authenticate())` and `checkBaseFunction(BaseFunction.ADMIN)`) is repeated in the `log`, `batchReindex`, and `mapPort` methods. This could be refactored into a reusable method

### **Inappropriate Naming**
- **`version` Method**:
  - The method name `version` is misleading because it retrieves more than just the application version (e.g., memory information). A more descriptive name like `getAppInfo` or `getVersionAndMemoryInfo` would be better.
- **`response` Variable**:
  - The variable `response` (a `JSONObject`) is reused in multiple methods without a specific context. Consider using more descriptive names like `versionInfo`, `logResponse`, or `reindexResponse`.

### **Primitive Obsession**
- **Query Parameters in `log` Method**:
  - The query parameters (`level`, `tag`, `message`, `limit`, `offset`) are passed as primitive types. Consider encapsulating them in a dedicated `LogRequest` object to improve readability and maintainability.

### **Lack of Error Handling**
- **`Appender` Check in `log` Method**:
  - The check for the `MEMORY` appender is done with a simple if condition. Consider throwing a more specific exception or logging the error before throwing the `ServerException`.

### **Hardcoded Strings**
- **"MEMORY" Appender Name**:
  - The appender name "MEMORY" is hardcoded in the `log` method. Consider using a constant or configuration value to avoid magic strings.
- **Exception Messages**:
  - Exception messages like "ServerError", "IndexingError", and "NetworkError" are hardcoded. These could be moved to constants or configuration files.

### **Potential NullPointerException**
- **`request.getServerPort()` in `mapPort` Method**:
  - The `request` object is not explicitly checked for null before calling `getServerPort()`. This could lead to a `NullPointerException` if `request` is null.

### **Unnecessary Complex Logic**
- **`NetworkUtil.mapTcpPort` Check in `mapPort` Method**:
  - The method throws an exception if the port mapping fails, but the error handling could be more descriptive or provide additional context.

### **Missing Documentation**
- **`mapPort` Method**:
  - The Javadoc for `mapPort` is minimal and does not explain the purpose or behavior of the method in detail.

### **Inconsistent Return Types**
- **`Response` Handling**:
  - While all methods return a `Response` object, the structure of the `JSONObject` being returned varies across methods. Consider standardizing the response structure for consistency.

Design Smells:
--------------------------------------------------
Here's an analysis of the provided Java code for potential design smells:

### 1. **Large Class**
   - The `AppResource` class handles multiple unrelated responsibilities:
     - Application version information.
     - Log retrieval and filtering.
     - Rebuilding the article index.
     - Port mapping functionality.
   - This violates the Single Responsibility Principle (SRP), as the class is managing too many unrelated tasks.

### 2. **God Object**
   - The class acts as a centralized point for diverse functionalities (e.g., logging, indexing, network port mapping).
   - It depends on multiple external utilities (`NetworkUtil`, `AppContext`, `MemoryAppender`, etc.), making it a potential God Object.

### 3. **Poor Abstraction**
   - The class directly interacts with low-level utilities (e.g., logging, memory management, network operations) without proper encapsulation.
   - Lack of abstraction for functionalities like logging retrieval or port mapping makes the code tightly coupled to specific implementations.

### 4. **Duplicate Code**
   - The authentication check (`if (!authenticate())`) and admin function verification (`checkBaseFunction(BaseFunction.ADMIN)`) are repeated in multiple methods.
   - This could lead to maintenance issues if authentication logic changes.

### 5. **Circular Dependencies**
   - No explicit circular dependencies are observed in this class, but the reliance on global instances (`AppContext.getInstance()`) could lead to indirect circular dependencies in a larger codebase.

### 6. **Tight Coupling**
   - The class is tightly coupled to specific implementations (e.g., `MemoryAppender`, `AppContext`, `NetworkUtil`).
   - This reduces flexibility and makes testing or swapping implementations difficult.
```

- There was also few cases of DeepSeek not identiying any smells in the code at all as seen below



```
=== Identified Smells (Gemini) ===

Code Smells:
--------------------------------------------------
**Potential Code Smells:**

* **Long method:** `list()` method contains over 20 lines of code.
* **Duplicated blocks:** The `for` loop contains duplicated code for creating and adding `JSONObject` items.
* **Inappropriate naming:** The `list()` method does not clearly indicate its purpose (listing locales).

Design Smells:
--------------------------------------------------
* **Large classes**: No large classes detected.
* **God objects**: No god objects detected.
* **Circular dependencies**: No circular dependencies detected.
* **Poor abstraction**: No poor abstraction detected.
--------------------------------------------------

=== Identified Smells (DeepSeek) ===

Code Smells:
--------------------------------------------------
None detected

Design Smells:
--------------------------------------------------
None detected
```

- Overall Gemini was the better model in identifying code and design smells.

Code Refactoring

Refactoring Results

The results for this task varied between the two models. **Gemini** held a slight edge over **DeepSeek**, being deemed better at refactoring in 60% of cases compared to 40%. Upon manual inspection of the final code, Gemini consistently produced cleaner and more readable outputs.

DeepSeek, on the other hand, sometimes failed to properly resolve the detected smell issues, leading to ineffective code refactoring in some test runs. While Gemini also occasionally failed to resolve certain issues, it rarely caused a reduction in metrics. This was not the case for DeepSeek, whose refactored code sometimes resulted in a decline in code quality metrics.

The difference in performance between the two LLMs could be attributed to the quality of the smell reports provided to each model. Gemini's concise reports, with fewer hallucinations and incorrect smells identified, likely contributed to its superior performance in refactoring tasks.

However, the choice of evaluation metrics—**Lines of Code (LOC)** and **Cyclomatic Complexity**—might have influenced the results. Further testing using additional or alternative metrics could provide a more comprehensive assessment.

Additionally, using models specifically tailored for code refactoring, rather than general-purpose models, might yield better results in future experiments.

Observations

- DeepSeek and Gemini performed near equally based on the number of times they were picked for refactoring.
- Manual inspection of code revealed that Gemini code was better.
- In most cases , both models provided usable refactored code
- Overall Gemini is the better model for the task.

General Insights

- Gemini performed better for both the tasks, however it was not always accurate hence cannot be used without manual oversight in real-world codebases.

- Both Models performed better at the smell detection task as compared to the refactoring task.

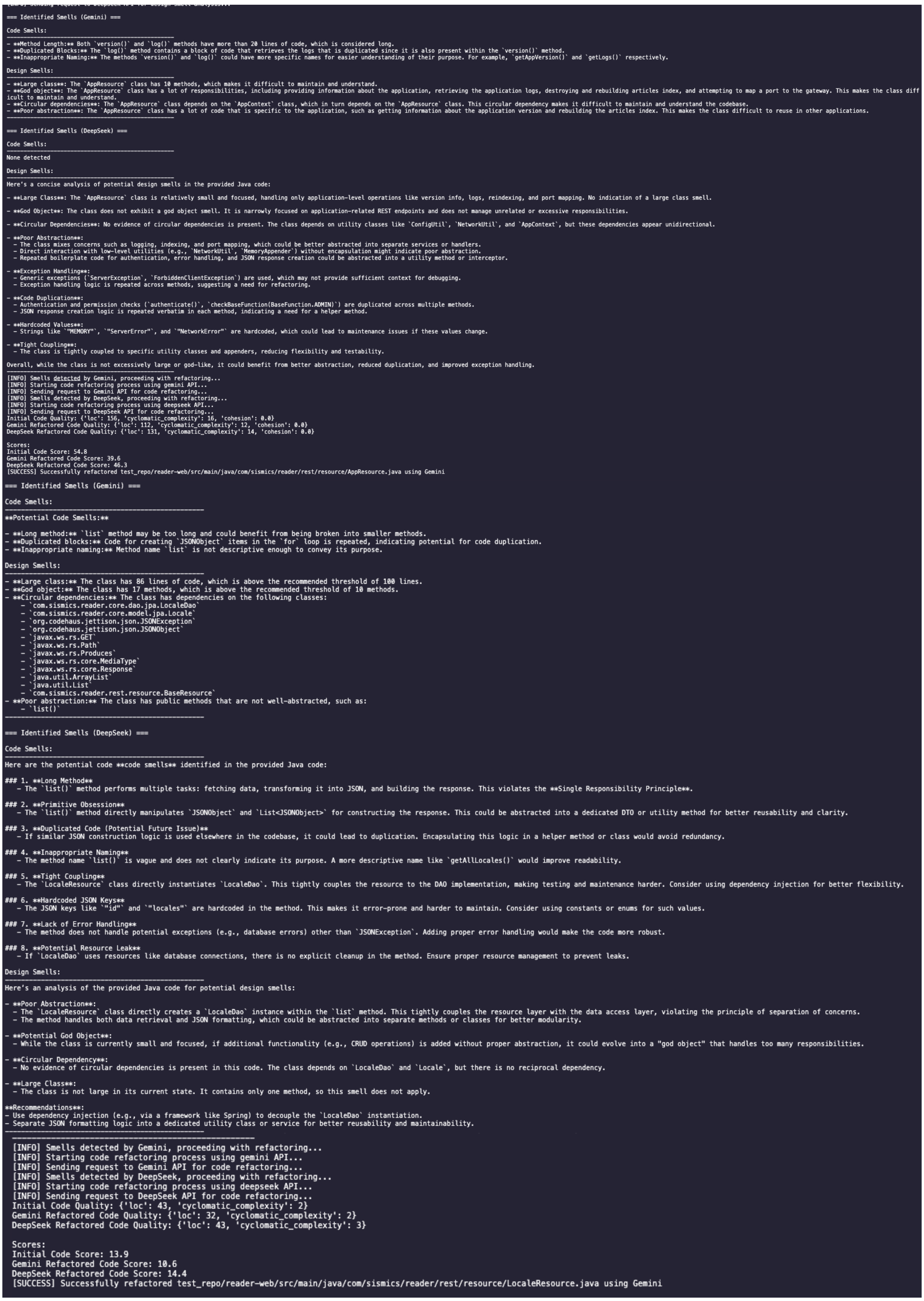
Potential Improvements

- Combining files with similar functionality to allow the API's to identify smells across files better.

- Providing Class Diagrams of the Codebase while identifying smells.

- Using a combinations of smells detected by multiple LLM's as context for the model refactoring the code.

Sample Output Images



```
[INFO] Sending request to DeepSeek API for design smell analysis...

=== Identified Smells (Gemini) ===

Code Smells:
--------------------------------------------------
- **Method Length:** Both `version()` and `log()` methods have more than 20 lines of code, which is considered long.
- **Duplicated Blocks:** The `log()` method contains a block of code that retrieves the logs that is duplicated since it is also present within the `version()` method.
- **Inappropriate Naming:** The methods `version()` and `log()` could have more specific names for easier understanding of their purpose. For example, `getAppVersion()` and `getLogs()` respectively.

Design Smells:
--------------------------------------------------
- **Large class**: The `AppResource` class has 10 methods, which makes it difficult to maintain and understand.
- **God object**: The `AppResource` class has a lot of responsibilities, including providing information about the application, retrieving the application logs, destroying and rebuilding articles index, and attempting to map a port to the gateway. This makes the class diff
icult to maintain and understand.
- **Circular dependencies**: The `AppResource` class depends on the `AppContext` class, which in turn depends on the `AppResource` class. This circular dependency makes it difficult to maintain and understand the codebase.
- **Poor abstraction**: The `AppResource` class has a lot of code that is specific to the application, such as getting information about the application version and rebuilding the articles index. This makes the class difficult to reuse in other applications.
--------------------------------------------------

=== Identified Smells (DeepSeek) ===

Code Smells:
--------------------------------------------------
None detected

Design Smells:
--------------------------------------------------
Here's a concise analysis of potential design smells in the provided Java code:

- **Large Class**: The `AppResource` class is relatively small and focused, handling only application-level operations like version info, logs, reindexing, and port mapping. No indication of a large class smell.

- **God Object**: The class does not exhibit a god object smell. It is narrowly focused on application-related REST endpoints and does not manage unrelated or excessive responsibilities.

- **Circular Dependencies**: No evidence of circular dependencies is present. The class depends on utility classes like `ConfigUtil`, `NetworkUtil`, and `AppContext`, but these dependencies appear unidirectional.

- **Poor Abstraction**:
  - The class mixes concerns such as logging, indexing, and port mapping, which could be better abstracted into separate services or handlers.
  - Direct interaction with low-level utilities (e.g., `NetworkUtil`, `MemoryAppender`) without encapsulation might indicate poor abstraction.
  - Repeated boilerplate code for authentication, error handling, and JSON response creation could be abstracted into a utility method or interceptor.

- **Exception Handling**:
  - Generic exceptions (`ServerException`, `ForbiddenClientException`) are used, which may not provide sufficient context for debugging.
  - Exception handling logic is repeated across methods, suggesting a need for refactoring.

- **Code Duplication**:
  - Authentication and permission checks (`authenticate()`, `checkBaseFunction(BaseFunction.ADMIN)`) are duplicated across multiple methods.
  - JSON response creation logic is repeated verbatim in each method, indicating a need for a helper method.

- **Hardcoded Values**:
  - Strings like `"MEMORY"`, `"ServerError"`, and `"NetworkError"` are hardcoded, which could lead to maintenance issues if these values change.

- **Tight Coupling**:
  - The class is tightly coupled to specific utility classes and appenders, reducing flexibility and testability.

Overall, while the class is not excessively large or god-like, it could benefit from better abstraction, reduced duplication, and improved exception handling.
--------------------------------------------------
[INFO] Smells detected by Gemini, proceeding with refactoring...
[INFO] Starting code refactoring process using gemini API...
[INFO] Sending request to Gemini API for code refactoring...
[INFO] Smells detected by DeepSeek, proceeding with refactoring...
[INFO] Starting code refactoring process using deepseek API...
[INFO] Sending request to DeepSeek API for code refactoring...
Initial Code Quality: {'loc': 156, 'cyclomatic_complexity': 16, 'cohesion': 0.0}
Gemini Refactored Code Quality: {'loc': 112, 'cyclomatic_complexity': 12, 'cohesion': 0.0}
DeepSeek Refactored Code Quality: {'loc': 131, 'cyclomatic_complexity': 14, 'cohesion': 0.0}

Scores:
Initial Code Score: 54.8
Gemini Refactored Code Score: 39.6
DeepSeek Refactored Code Score: 46.3
[SUCCESS] Successfully refactored test_repo/reader-web/src/main/java/com/sismics/reader/rest/resource/AppResource.java using Gemini

=== Identified Smells (Gemini) ===

Code Smells:
--------------------------------------------------
**Potential Code Smells:**

- **Long method:** `list` method may be too long and could benefit from being broken into smaller methods.
- **Duplicated blocks:** Code for creating `JSONObject` items in the `for` loop is repeated, indicating potential for code duplication.
- **Inappropriate naming:** Method name `list` is not descriptive enough to convey its purpose.

Design Smells:
--------------------------------------------------
- **Large class:** The class has 86 lines of code, which is above the recommended threshold of 100 lines.
- **God object:** The class has 17 methods, which is above the recommended threshold of 10 methods.
- **Circular dependencies:** The class has dependencies on the following classes:
    - `com.sismics.reader.core.dao.jpa.LocaleDao`
    - `com.sismics.reader.core.model.jpa.Locale`
    - `org.codehaus.jettison.json.JSONException`
    - `org.codehaus.jettison.json.JSONObject`
    - `javax.ws.rs.GET`
    - `javax.ws.rs.Path`
    - `javax.ws.rs.Produces`
    - `javax.ws.rs.core.MediaType`
    - `javax.ws.rs.core.Response`
    - `java.util.ArrayList`
    - `java.util.List`
    - `com.sismics.reader.rest.resource.BaseResource`
- **Poor abstraction:** The class has public methods that are not well-abstracted, such as:
    - `list()`
--------------------------------------------------

=== Identified Smells (DeepSeek) ===

Code Smells:
--------------------------------------------------
Here are the potential code **code smells** identified in the provided Java code:

### 1. **Long Method**
   - The `list()` method performs multiple tasks: fetching data, transforming it into JSON, and building the response. This violates the **Single Responsibility Principle**.

### 2. **Primitive Obsession**
   - The `list()` method directly manipulates `JSONObject` and `List<JSONObject>` for constructing the response. This could be abstracted into a dedicated DTO or utility method for better reusability and clarity.

### 3. **Duplicated Code (Potential Future Issue)**
   - If similar JSON construction logic is used elsewhere in the codebase, it could lead to duplication. Encapsulating this logic in a helper method or class would avoid redundancy.

### 4. **Inappropriate Naming**
   - The method name `list()` is vague and does not clearly indicate its purpose. A more descriptive name like `getAllLocales()` would improve readability.

### 5. **Tight Coupling**
   - The `LocaleResource` class directly instantiates `LocaleDao`. This tightly couples the resource to the DAO implementation, making testing and maintenance harder. Consider using dependency injection for better flexibility.

### 6. **Hardcoded JSON Keys**
   - The JSON keys like `"id"` and `"locales"` are hardcoded in the method. This makes it error-prone and harder to maintain. Consider using constants or enums for such values.

### 7. **Lack of Error Handling**
   - The method does not handle potential exceptions (e.g., database errors) other than `JSONException`. Adding proper error handling would make the code more robust.

### 8. **Potential Resource Leak**
   - If `LocaleDao` uses resources like database connections, there is no explicit cleanup in the method. Ensure proper resource management to prevent leaks.

Design Smells:
--------------------------------------------------
Here's an analysis of the provided Java code for potential design smells:

- **Poor Abstraction**:
  - The `LocaleResource` class directly creates a `LocaleDao` instance within the `list` method. This tightly couples the resource layer with the data access layer, violating the principle of separation of concerns.
  - The method handles both data retrieval and JSON formatting, which could be abstracted into separate methods or classes for better modularity.

- **Potential God Object**:
  - While the class is currently small and focused, if additional functionality (e.g., CRUD operations) is added without proper abstraction, it could evolve into a "god object" that handles too many responsibilities.

- **Circular Dependency**:
  - No evidence of circular dependencies is present in this code. The class depends on `LocaleDao` and `Locale`, but there is no reciprocal dependency.

- **Large Class**:
  - The class is not large in its current state. It contains only one method, so this smell does not apply.

**Recommendations**:
- Use dependency injection (e.g., via a framework like Spring) to decouple the `LocaleDao` instantiation.
- Separate JSON formatting logic into a dedicated utility class or service for better reusability and maintainability.
--------------------------------------------------
 --------------------------------------------------
 [INFO] Smells detected by Gemini, proceeding with refactoring...
 [INFO] Starting code refactoring process using gemini API...
 [INFO] Sending request to Gemini API for code refactoring...
 [INFO] Smells detected by DeepSeek, proceeding with refactoring...
 [INFO] Starting code refactoring process using deepseek API...
 [INFO] Sending request to DeepSeek API for code refactoring...
 Initial Code Quality: {'loc': 43, 'cyclomatic_complexity': 2}
 Gemini Refactored Code Quality: {'loc': 32, 'cyclomatic_complexity': 2}
 DeepSeek Refactored Code Quality: {'loc': 43, 'cyclomatic_complexity': 3}

 Scores:
 Initial Code Score: 13.9
 Gemini Refactored Code Score: 10.6
 DeepSeek Refactored Code Score: 14.4
 [SUCCESS] Successfully refactored test_repo/reader-web/src/main/java/com/sismics/reader/rest/resource/LocaleResource.java using Gemini
```