That's an excellent idea! This approach can significantly improve the flexibility and maintainability of your framework. Here's how you can implement this concept:

**1. Enhance ConfigService**

* Modify ConfigService to accept a key (e.g., 'SRC\_FILE\_1') as an argument.
* Implement the logic to retrieve the corresponding file/database metadata (input file path, output file path, validation rules, transformation rules, etc.) based on the provided key.

Python

class ConfigService:  
 def get\_file\_metadata(self, file\_key: str) -> dict:  
 """  
 Fetches file/database metadata based on the provided key.  
  
 Args:  
 file\_key: The key identifying the file/database metadata.  
  
 Returns:  
 A dictionary containing the metadata (input\_file\_path, output\_file\_path,   
 validation\_rules, transformation\_rules, etc.).  
 """  
 # ... implementation to fetch metadata based on file\_key ...

**2. Modify ProcessFileUseCase**

* Adjust ProcessFileUseCase to accept the file\_key as an argument.
* Use the ConfigService to retrieve the metadata for the specified file.
* Utilize the retrieved metadata to configure the reader, writer, transformer, and validator components.

Python

class ProcessFileUseCase:  
 def \_\_init\_\_(self, config\_service: ConfigService,   
 csv\_reader: CSVReader,   
 data\_transformer: DataTransformer,   
 data\_validator: DataValidator,   
 stats\_collector: StatsCollector,   
 logger: Logger,   
 csv\_writer: CSVWriter):  
 self.config\_service = config\_service  
 self.csv\_reader = csv\_reader  
 self.data\_transformer = data\_transformer  
 self.data\_validator = data\_validator  
 self.stats\_collector = stats\_collector  
 self.logger = logger  
 self.csv\_writer = csv\_writer  
  
 def execute(self, file\_key: str) -> None:  
 file\_metadata = self.config\_service.get\_file\_metadata(file\_key)   
 input\_file\_path = file\_metadata['input\_file\_path']  
 output\_file\_path = file\_metadata['output\_file\_path']  
 # ...   
 # Configure data\_transformer and data\_validator   
 # using rules from file\_metadata   
 # ...  
  
 data = self.csv\_reader.read(input\_file\_path)  
 if self.data\_validator.validate(data):  
 transformed\_data = self.data\_transformer.transform(data)  
 stats = self.stats\_collector.collect\_stats(data)  
 self.logger.log(f"Processed {len(data)} rows. Statistics: {stats}")  
 self.csv\_writer.write(transformed\_data, output\_file\_path)  
 else:  
 self.logger.log("Data validation failed.", level='ERROR')

**3. Modify main.py**

* Pass the file\_key to the execute() method of the ProcessFileUseCase.

Python

def main():  
 db\_connection = # ... establish database connection ...  
 config\_service = DatabaseConfigService(db\_connection)   
 # ... other component initializations ...  
 use\_case = ProcessFileUseCase(config\_service,   
 csv\_reader,   
 data\_transformer,   
 data\_validator,   
 stats\_collector,   
 logger,   
 csv\_writer)  
 use\_case.execute("SRC\_FILE\_1")

**Benefits:**

* **Centralized Configuration:** All file-specific metadata is managed centrally within the ConfigService.
* **Flexibility:** Easily add, modify, or remove file processing configurations by updating the metadata in the configuration source.
* **Maintainability:** Improved maintainability by separating file-specific configurations from the core business logic.
* **Extensibility:** Easily extend the framework to handle different file processing scenarios by simply adding new entries to the metadata.

By incorporating this file\_key mechanism, you can significantly enhance the flexibility, maintainability, and scalability of your Hexagonal Architecture-based file processing framework.

I hope this refined approach provides a more robust and adaptable solution for your project!