You're absolutely right! The previous example missed a crucial aspect: **fetching validation and transformation rules from a configuration source (like a database or a configuration file).**

Here's how we can incorporate this into our architecture:

**1. infrastructure/config/config\_service.py**

Python

from abc import ABC, abstractmethod  
  
class ConfigService(ABC):  
 @abstractmethod  
 def get\_validation\_rules(self) -> dict:  
 """Fetches validation rules from the configuration source."""  
 pass  
  
 @abstractmethod  
 def get\_transformation\_rules(self) -> dict:  
 """Fetches transformation rules from the configuration source."""  
 pass  
  
class DatabaseConfigService(ConfigService):  
 def \_\_init\_\_(self, db\_connection):   
 self.db\_connection = db\_connection   
  
 def get\_validation\_rules(self) -> dict:  
 # Logic to fetch validation rules from the database  
 pass  
  
 def get\_transformation\_rules(self) -> dict:  
 # Logic to fetch transformation rules from the database  
 pass  
  
class FileConfigService(ConfigService):  
 def \_\_init\_\_(self, config\_file\_path):  
 self.config\_file\_path = config\_file\_path   
  
 def get\_validation\_rules(self) -> dict:  
 # Logic to read validation rules from the configuration file  
 pass  
  
 def get\_transformation\_rules(self) -> dict:  
 # Logic to read transformation rules from the configuration file  
 pass

**2. domain/services/data\_transformer.py**

Python

class DataTransformer:  
 def \_\_init\_\_(self, config\_service: ConfigService):  
 self.config\_service = config\_service  
 self.transformation\_rules = self.config\_service.get\_transformation\_rules()   
  
 def transform(self, data: list[dict]) -> list[dict]:  
 # ... implementation for data transformation   
 # using the fetched transformation\_rules   
 pass

**3. domain/services/data\_validator.py**

Python

class DataValidator:  
 def \_\_init\_\_(self, config\_service: ConfigService):  
 self.config\_service = config\_service  
 self.validation\_rules = self.config\_service.get\_validation\_rules()  
  
 def validate(self, data: list[dict]) -> bool:  
 # ... implementation for data validation   
 # using the fetched validation\_rules   
 pass

**4. application/use\_cases/process\_file\_use\_case.py**

Python

from application.use\_cases.process\_file\_use\_case import ProcessFileUseCase  
  
class ProcessFileUseCase:  
 def \_\_init\_\_(self, csv\_reader: CSVReader, data\_transformer: DataTransformer,   
 data\_validator: DataValidator, csv\_writer: CSVWriter):  
 self.csv\_reader = csv\_reader  
 self.data\_transformer = data\_transformer  
 self.data\_validator = data\_validator  
 self.csv\_writer = csv\_writer  
  
 def execute(self, input\_file\_path: str, output\_file\_path: str) -> None:  
 data = self.csv\_reader.read(input\_file\_path)  
 if self.data\_validator.validate(data):  
 transformed\_data = self.data\_transformer.transform(data)  
 self.csv\_writer.write(transformed\_data, output\_file\_path)  
 else:  
 # Handle validation errors   
 pass

**5. application/main.py**

Python

from application.use\_cases.process\_file\_use\_case import ProcessFileUseCase  
from infrastructure.config.config\_service import DatabaseConfigService   
  
def main():  
 db\_connection = # ... establish database connection ...  
 config\_service = DatabaseConfigService(db\_connection)   
 csv\_reader = CSVReader()  
 data\_transformer = DataTransformer(config\_service)  
 data\_validator = DataValidator(config\_service)  
 csv\_writer = CSVWriter()  
 use\_case = ProcessFileUseCase(csv\_reader, data\_transformer, data\_validator, csv\_writer)  
 use\_case.execute("input.csv", "output.csv")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

By introducing the ConfigService abstraction, we've made our architecture more flexible. You can easily switch between different configuration sources (database, file) by simply changing the implementation of the ConfigService class.

This revised example demonstrates how to fetch validation and transformation rules from a configuration source, making your file processing framework more dynamic and adaptable.