You're absolutely right! We need to incorporate **stats gathering** and **logging** into our architecture. Here's how we can integrate these components:

**1. domain/services/stats\_collector.py**

Python

class StatsCollector:  
 def collect\_stats(self, data: list[dict]) -> dict:  
 """Collects statistics about the data (e.g., number of rows, number of unique values, etc.)."""  
 # ... implementation for collecting statistics ...  
 return stats\_data

**2. adapters/logger/logger.py**

Python

from abc import ABC, abstractmethod  
  
class Logger(ABC):  
 @abstractmethod  
 def log(self, message: str, level: str = 'INFO') -> None:  
 """Logs a message with the specified log level."""  
 pass  
  
class FileLogger(Logger):  
 def \_\_init\_\_(self, log\_file\_path: str):  
 self.log\_file\_path = log\_file\_path  
  
 def log(self, message: str, level: str = 'INFO') -> None:  
 # ... implementation for logging to a file ...  
  
class DatabaseLogger(Logger):  
 def \_\_init\_\_(self, db\_connection):  
 self.db\_connection = db\_connection  
  
 def log(self, message: str, level: str = 'INFO') -> None:  
 # ... implementation for logging to a database ...

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

Python

class ProcessFileUseCase:  
 def \_\_init\_\_(self, csv\_reader: CSVReader, data\_transformer: DataTransformer,   
 data\_validator: DataValidator, stats\_collector: StatsCollector,   
 logger: Logger, csv\_writer: CSVWriter):  
 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, input\_file\_path: str, output\_file\_path: str) -> None:  
 try:  
 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')  
 except Exception as e:  
 self.logger.log(f"An error occurred: {str(e)}", level='ERROR')

**4. application/main.py**

Python

from application.use\_cases.process\_file\_use\_case import ProcessFileUseCase  
from infrastructure.config.config\_service import DatabaseConfigService  
from adapters.logger.file\_logger import FileLogger  
  
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)  
 stats\_collector = StatsCollector()   
 logger = FileLogger("process\_log.txt")   
 csv\_writer = CSVWriter()  
 use\_case = ProcessFileUseCase(csv\_reader, data\_transformer, data\_validator,   
 stats\_collector, logger, csv\_writer)  
 use\_case.execute("input.csv", "output.csv")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

By incorporating StatsCollector and Logger into our architecture, we now have the ability to gather statistics about the processed data and log events throughout the execution of the file processing pipeline.

This enhanced architecture provides a more comprehensive and informative solution for your file processing framework.