CS-499 Capstone Project: Design Documentation

Contents

[Introduction 1](#_Toc199084522)

[Scope 1](#_Toc199084523)

[1. Functional Specification 1](#_Toc199084524)

[2. Security Considerations 2](#_Toc199084525)

[3. Design Specification 3](#_Toc199084526)

# Introduction

This document acts as both a functional specification and a design specification for a solution aimed at assisting system administrators in migrating CSV data into MongoDB. The goal of this solution is to provide a tool that automates the import process while offering flexibility for use in a variety of data sources, some data validation, and error handling. This document aims to define the core functionality, basic requirements, data import methodologies, and security considerations necessary to support the migration solution. As this specification combines both the functional specifications and the design specification, it will be updated throughout the development process.

# Scope

This scope of this document will be limited to describing functionality and technical design details relevant to the implementation of the solution. Within the sections describing the functionality, a brief overview of the requirements that the solution will address with its functionality will be provided. Each element of the functionality will be described individually. The sections of this document describing the design specification will outline the technical details for implementation of the solution. This is intended to aid in both development and future maintenance and added features.

# 1. Functional Specification

**1.1 Overview**

This section outlines the core functional requirements of the CSV-to-MongoDB migration tool. The functionality described here ensures that system administrators can use the tool flexibly across multiple environments and data sources while maintaining reliability, performance, and ease of troubleshooting.

**1.2 CSV Input Handling**

* The tool shall dynamically read CSV headers at runtime, regardless of the structure or contents of the data rows.
* The solution shall not rely on predefined column mappings and must adapt to any schema present in the source CSV file.
* The tool shall gracefully handle null or missing data in any column without failing or corrupting the document structure during insertion.

**1.3 MongoDB Connectivity**

* The tool shall support connections to multiple MongoDB instances.
* System administrators shall be able to configure and switch between various MongoDB URIs and target databases/collections within the application interface or configuration file.
* The tool shall support inserting records as individual documents, ensuring atomicity and adherence to MongoDB’s best practices for document storage.

**1.4 Data Storage and Processing**

* The parsed CSV data shall be stored in an internal data structure optimized for rapid transformation and efficient insertion into MongoDB (e.g., dictionaries or in-memory collections).
* Each CSV row shall correspond to a single MongoDB document, structured based on the CSV headers.

**1.5 Error Handling and Logging**

* The solution shall detect and handle the following types of errors:
  + Errors during CSV parsing (e.g., malformed rows, encoding issues).
  + Errors during MongoDB operations (e.g., connection failures, validation errors).
* All errors shall be logged with detailed contextual information, including timestamps, error type, and affected records.
* The system shall provide a logging mechanism that outputs to a configurable log file, enabling troubleshooting and auditability for system administrators.

# 2. Security Considerations

To protect sensitive infrastructure and data during CSV-to-MongoDB migrations, the tool shall incorporate the following security features:

* **Credential Protection**: MongoDB connection strings containing usernames and passwords must be encrypted when stored and masked in logs or output messages.
* **Role-Based Access**: The tool should only allow access to MongoDB instances using users with the least privileges necessary to perform insert operations.
* **Validation and Sanitization**: Input data from CSV files must be validated and sanitized to prevent injection attacks or the insertion of unexpected or malformed data structures.
* **TLS/SSL Support**: Connections to MongoDB should support TLS/SSL encryption to protect data in transit.
* **Audit Trail**: All user actions and database interactions should be logged securely to provide an audit trail for administrative review.
* **File Access Control**: The application should restrict access to the CSV files and configuration files to authorized users only, using file system permissions.

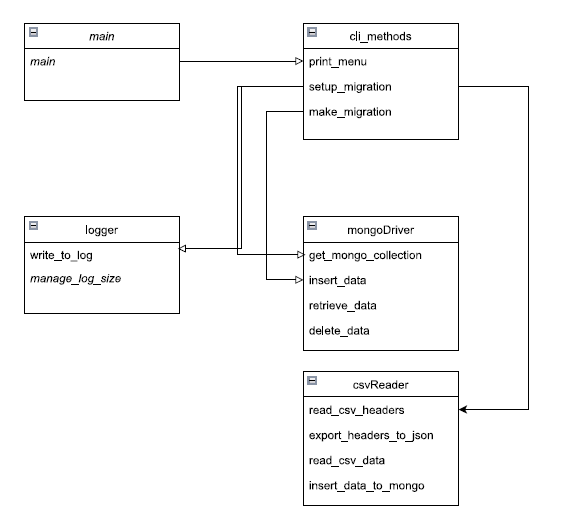
# 3. Design Specification

**Note:** The design specification will be updated as implementation of each of the technical details satisfying the requirements. Any elements marked as optional are to be considered secondary considerations when implementing the solution, and are only to be included if the primary technical solution implements it natively.

**3.1 System Architecture**

**High-Level Architecture Diagram**

Below is diagram illustrating the interactions between the different methods used to build the program.



**Deployment Model**

Porter is designed to be deployed as a command line interface utility for system administrators. While the command line interface client can be run locally, it would more commonly be made available on the source or destination server. This would leverage some of the built in security considerations governing access to the source or destination data servers for the utility.

Cross-platform compatibility between Windows and Linux servers is established since the solution is written purely in Python. As long as Python is available on the server running the utility and that server can be accessed administrators will be able to interact with the application.

**3.2 CSV Input Module**

**3.2.1 Header Parsing**

**Auto-detection of headers on the first row is i**mplemented in csvReader.read\_csv\_headers(), which reads and returns the first row of the CSV as headers. This is used to build the migration file. **To allow for dynamic schema mapping using dictionaries, h**eaders are read dynamically and exported to a JSON structure under a unique key per file (export\_headers\_to\_json()), supporting schema mapping via key association.

**3.2.2 Data Normalization**

The handling of null values is accomplished by inserting null or omitting keys. The built in class csv.DictReader is used, which automatically parses empty fields as empty strings. Explicit null conversion was omitted to allow for maximum flexibility in destination MongoDB documents. The rationale for this is data sanitization is out of scope for the intended application of this utility. Similarly, whitespace trimming is not currently implemented.

**3.2.3 Performance Considerations**

During development a “light touch” approach to performance improvements was taken. System administrators are expected to use their best judgement when handling data large data migrations. The use of buffered reading for large files is accomplished by virtue of the fact that Python's csv module inherently supports buffered reading.

Optional chunking of large datasets is not yet implemented. All rows are currently read into memory at once in read\_csv\_data().Future enhancements could introduce faster insertion indexing or allowing system administrators to stage the data migration in steps.

**3.3 MongoDB Integration Module**

**3.3.1 Connection Management**

Connections to MongoDB URIs via are handled by the user. The connection string is stored in a user-generated JSON config during the setup\_migration() phase.

Pooling and timeout handling for stability are left at the defaults from pymongo.MongoClient. These are defaulted so that connection pooling is implicitly handled but not explicitly customized. The rationale for this decision is that it is not expected that a single migration will be handling multiple data stores, rather that individual migration files will be generated.

**3.3.2 Data Insertion Strategy**

The principle of one document per row was used for the data insertion strategy. This was determined based on the earlier principal that the system administrator would not be responsible for handling data based decisions. Each row is read into a dictionary and inserted as a separate MongoDB document. For this reason, validation against expected schema is not yet provided. However, future improvements could involve schema documents provided by data engineers for the purpose of checking during the migration.

**3.4 Data Storage and Transformation**

**3.4.1 In-Memory Data Structure**

Use of Python dictionaries (or equivalent) keyed by headers: Implemented via csv.DictReader, which produces a list of dictionaries.

Temporary storage with cleanup after insert: Data is held temporarily in memory during make\_migration() but is not persisted or cached; cleanup occurs naturally **through scope.**

**3.4.2 Transformation Hooks**

Optional pre-insert transformation function (user-defined): Not yet implemented. Hook points can be added before calling insert\_data() or insert\_many().

Type coercion (e.g., convert strings to dates/numbers where possible): Not present in the current implementation, but the design allows for adding this within the CSV processing logic.

**3.5 Error Handling and Logging**

**3.5.1 Error Detection**

Structured exception handling for file I/O and DB operations in key operations such as (file reads, JSON parsing, CSV processing, and MongoDB inserts) are wrapped in try-except blocks that detect and gracefully handle runtime exceptions. When an exception occurs, the message is logged using the write\_to\_log() function from logger.py, allowing consistent and centralized handling.

**3.5.2 Logging Architecture**

The utility employs a centralized logger with various logging levels (INFO, WARN, ERROR). The logger.py module defines a centralized logging function write\_to\_log(), supporting levels such as DEBUG, INFO, WARNING, ERROR, and CRITICAL. This allows developers and users to control the granularity of log data. The information is output to file with rotation and console output left included for CLI methods.

All logging is written to a single specified log file. A custom log rotation mechanism is implemented in manage\_log\_size(), which truncates entries older than 6 months when the log file exceeds 5MB.

Logs include timestamps and severity levels. While the current implementation does not explicitly log file name, row number, or collection name, these can be embedded in the message passed to write\_to\_log() (e.g., "Row 42 in file mydata.csv failed to insert into 'bids' collection").

**3.6 Configuration Management**

**3.6.1 Configuration File Schema**

JSON or YAML format for user-defined settings are implemented using JSON. setup\_migration() creates a config JSON file with paths, file count, and connection string. The fields, MongoDB URIs, target collections, logging level, input file path, batch size are at the time of this initial publication only partially implemented. MongoDB URI and input file paths are stored. Logging level and batch size are not yet handled.

**3.6.2 Environment Variable Support**

For use in a production environment, environmental variable support will need to be added. The implementation is intended to showcase the application in a development state.

**3.7 Security Design**

**3.7.1 Connection Security**

TLS/SSL support for MongoDB connections are supported implicitly through MongoDB URI configuration, but not enforced or checked. Masked credentials in logs are not implemented, since the logging module is fully customizable, messages should be sanitized or modified prior to implementation in a production environment.

**3.7.2 Access Control**

The principle of enforced use of MongoDB users with least privilege is not enforced in code. Depends on the credentials provided in the MongoDB URI. This decision is intentionally left up to system administrators.

**3.7.3 File Permissions and Safe Logging**

Restrict access to CSV and config files is not enforced programmatically. Follow your site’s documentation for best practices.

**3.8 User Interface**

**3.8.1 CLI Design**

Command-line flags for quick control (e.g., --input, --mongo-uri): Not implemented. The CLI is menu-driven using input() calls.

Help and error messages for usability: Some basic error messages and prompts are included, but no --help flag or CLI parser like argparse is used.