International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 13 Issue 6 ǁ June 2025 ǁ PP. 124-133

Mock Data Generator and Seeder

Dr. Nilesh Mali, Vedant Ghodekar, Prasanna Baviskar, Piyush Mali, Subham Patro
Ajeenkya D. Y. Patil School of Engineering, Pune – 412105

Abstract—*The Mock Database Generator and Seeder project aims to provide a streamlined and automated solution for generating, validating, and seeding mock data into databases, particularly for development and testing environments. The tool is designed to handle complex database schemas by utilizing a configuration file that specifies data structure, types, and constraints, ensuring realistic and representative mock data. It supports both data generation from scratch and the integration of pre-existing datasets, such as external SQLite mock data, to enhance the testing process. With the ability to connect to a variety of database systems, including PostgreSQL, the tool automates the seeding process, reducing the manual overhead involved in setting up test databases. This project is particularly beneficial for developers, testers, and data engineers who require reliable, consistent mock data for system testing, performance benchmarking, or data validation. By automating these processes, the Mock Database Generator and Seeder significantly reduces time, effort, and errors, making it a vital tool for efficient software development cycles.*
Index Terms—*Mock Data Generation, Database Seeding Automation, Data Validation and Testing*

Date of Submission: 01-06-2025 Date of acceptance: 10-06-2025

I. INTRODUCTION

The Mock Database Generator and Seeder project is a response to the lingering problem of the generation and management of realistic datasets, which have become important in present-day software development and the testing sphere. In the current software world, testing applications in-depth is tremendously crucial for insuring reliability, security, and performance. Although there are other problems like data privacy, limited access to real data, and the size and complexity of production databases, it is most impractical for tests to work directly on production data. The data privacy concerns are aggravated by the existence of extant regulations-fast proliferating in modern times, such as GDPR, CCPA-which govern the use of sensitive data. Additionally, testing with production data may lead to the risk of exposing private information of the user or breach compliance. That is why there is an increasing need for an alternative: safe, scalable, and automated mock data generation for tests.

Mock Database Generator and Seeder do provide a solution to the problem by automating the initial parts of generating, validating, and seeding mock data into databases such as PostgreSQL with little manual intervention. It empowers developers, testers, and data engineers to create and manage realistic test datasets in minutes that resemble the actual data in the sense of structure, relationships, and constraints. It greatly reduces time-consuming manual data entry, but above all gives the added benefit of having minimal scope for human error and scalable operation.

Mock data creation is not casually referred to as random data creation just for testing; it traditionally reflects how this data would behave in production scenarios, which is important for the testing of performance, integrity, and functions of a software application under realistic conditions. Mock data has to be created with a good degree of fidelity in mind, representing a wide range of real-world situations including variable user inputs, edge cases, and complicated data relationships. Realism is the cornerstone of these tests, and without it, issues that might arise in production can be missed. The Mock Database Generator and Seeder advocate this by allowing their users to define their configurations in detail and to assure that the mock data complies with the same constraints, patterns, and relations as the production data so that the test is thorough and credible. The infrastructure enables developers and testers to create realistic data responsive to real-world scenarios, allowing themselves to simulate user behavior on one hand, and on the other, load the system up to help validate performance plus ensure the system handles different data structures, which is popular for isolating potential bottlenecks or bugs before production.

What makes this mock data generator unique is its flexibility in defining the structure and characteristics of mock data. Users can specify schema, data types, relationships, and constraints within the data, making it very close to real business scenarios. This is the reason why this tool fits all kinds of verticals - from small applications having simple data models to large enterprise systems with complex database structures. Configuring these aspects makes this project worth its salt, owing to the datasets generated fitting customized specifications for any testing scenarios, equipping the developer's toolbox very nicely with this versatile tool.

Along with mock data generation from scratch, there is also an ability to incorporate third-party data sources, such as pre-existing SQLite mock data assets. This is another great means of ensuring increased versatility, enabling one to use previously developed mock datasets and thus generate timesaving mock data that's just that much realer. Regardless of whether the data is drawn from an existing dataset or generated live in real-time, the Mock Database Generator and Seeder guarantees the generated data is entirely appropriate for the given test scenario, and datasets produced are extremely representative and closer to real-world applications.

Ultimately, the Mock Database Generator and Seeder addresses a pertinent need in software development and testing: the demand to generate and manage mock data automatically, flexibly, and scalably. Thus, it enables developing projects to generate representative datasets for effective testing, thus speeding up the development cycle, decreasing errors, and improving the overall quality and reliability of the software being developed. This software project will allow the generation of ready-test data that closely reflects real-world situations, thereby now allowing teams to conduct clearer testing in identifying possible glitches at an earlier stage of development and confidently release much faster. The able-bodied ability to automate creation with consistent reliable data makes it particularly unique and this makes it indispensable in accelerating the work flow of developing software products.

II. BACKGROUND

A. Mock Data Generation

Mock data generation has gained tremendous importance in testing and validation processes for modern software development [1]. It allows one to simulate certain scenarios that would not warrant the use of actual production data to evaluate the developed software. Automated testing, performance evaluation, or API development may sometimes require using real data, where security, privacy, or unavailability may pose challenges [10]. Existing methods of mock data generation span manual creation, rule-based generation, and tools operating in a programmed manner, generating mock data based on prescribed schemas and structures [14]. Various implementations lack adequate flexibility to generate exhaustive sets of realistic and scalable test datasets [20].

B. Data Seeding

Data seeding is a technique where an application database is preloaded with initial data for testing and development [1]. The idea of seeding is to set up the conditions under which it is believed an application should run as expected before it is finally deployed [10]. The testing frameworks apply this in order that tests can depend on any specific datasets in validating functional requirements [14]. Traditional means of seeding involve static datasets that were inserted by hand into the database; this method can sometimes be quite tedious and susceptible to human error [20]. For example, automated seeding solutions create structured data in a more dynamic manner to cover a variety of testing cases and can thus improve testing even more [1]. A well-built seeder also promotes reproducibility to enable development teams to test software behaviors under all possible conditions [10].

C. Data Validation

Data validation and testing are crucial for ensuring the accuracy, consistency, and reliability of mock data used in software applications [10]. They involve a multiple-step process that includes data validation, schema validation, constraint enforcement, and data integrity checks [1]. Schema validation ensures the mock data fits the expected structure, while constraint enforcement maintains business logic parameters [14]. Data integrity checks ensure logical consistency and avoid duplicate records or missing references [20].Mock data should be diverse to cover more scenarios, especially in automated software testing [1]. Validation tools blended with the mock data generation framework can improve performance and security testing reliability [14]. However, challenges remain regarding realism, scale, and efficiency [10]. Large-scale applications require large amounts of mock data to represent realistic distributions without burdening computational functionality [1]. Automated validation techniques improve test setup reliability and reduce company overhead, especially in terms of application defects [14].

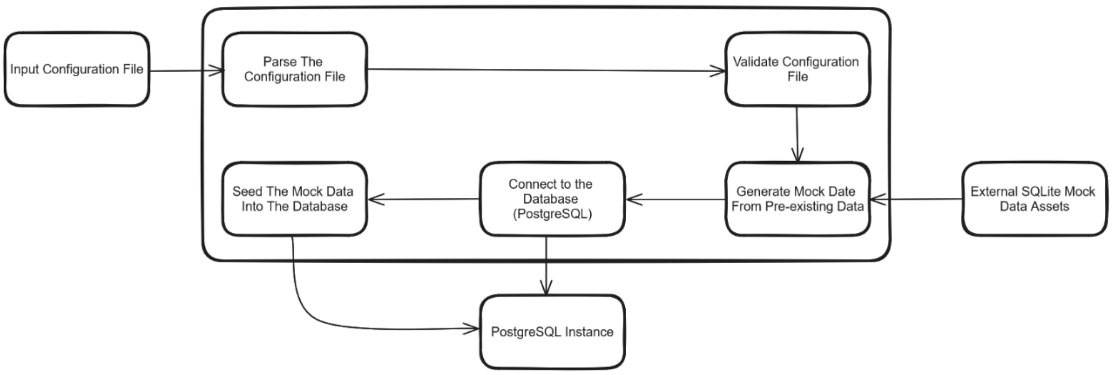
D. Database Schema Validation

Database schema validation is a critical aspect of mock data generation and seeding for various database types [5]. A well-defined schema enhances data integrity, linking those schemas, and making sure consistency is attained, which prevents errors resulting from invalid data structures [1]. It checks that the mock data also follows certain rules in conformity with table structures predefined, data types used, primary keys, foreign keys, as well as uniqueness on some values set [10]. Validation mechanisms integrated with mock data generation frameworks enhance software reliability by simulating realistic database conditions without needing to be provided with actual production data [14]. Combined with schema validation within mock data generation, developers would be producing realistic datasets very similar to those found within production, bringing on improvements to the effectiveness of database-driven testing for software [5].

```yaml
version: 1

database:
  provider: postgres
  name: test
  host: localhost
  port: 5432
  username: postgres
  password: postgres

files:
  path: /home/vedant/Workspace/mock/testdata/dataFiles
  files:
    - names.sqlite
    - surname.sqlite

tables:
  - name: users
    count: 20000
    columns:
      - name: name
        type: string
        from: names.names.name
      - name: surname
        type: string
        from: surname.surname.surname
```

Database Connection through YAML file

III. PROPOSED APPROACH ARCHITECTURE



The following section discusses the proposed architecture for a command-line interface tool to automate the creation, validation, and seeding of mock data in a PostgreSQL database. This automated flow is started by the Input Configuration File, which is a central document that describes the structure, constraints, and other details for how to generate mock data. The config file describes the database schema, including tables, fields, data types, relationships between them, and constraints such as primary keys, foreign keys, and uniqueness conditions. The config file could also include database connection settings, predefined constraints on data, and external data sources that the user might want to leverage to add additional realistic features to the generated mock data. Only then, once the configuration file has been specified, does the system step into the parsing phase where the tool reads and interprets the file to extract meaningful information. The highlighted parsing would include the identification of the underlying schema of the database, the relations among the records under consideration, and the constraints the mock data needs to abide by so that it would represent a real-world scenario.

Then follows the validation process, wherein checks are carried out to ensure that during the construction of the configuration file, the file is adhered to, according to certain predefined guidance and formats. This particular step also serves the ultimate aim of a technician by making sure no errors come that are going to propagate into later steps in the process. The validation mechanism will look into checking the inconsistencies and missing fields as well as whether the data type has been set out appropriately in the schema definition and whether the logical definition stands to what they set forth. If any discrepancies are found within the process, then properly identified error messages will be sent out to the user, calling for the areas that need amendments and why. Following validation of the configuration file, the tool will advance into mocked data generation, wherein fake data is produced according to the specifications in the input file.

In addition, the tool allows external SQLite mock data assets to be integrated so a user can enhance datasets generated or rely upon existing structured data in order to provide consistency with a historical or production-like dataset. A connection with the PostgreSQL database is established, once the mock data is generated, using the credentials and connection parameters as specified in the configuration file. A CLI tool assures that an authenticated session to create a connection is safe and stable in all respects and allows seamless interaction with the database system by offering the intelligence for authentication and session management to create seamless interaction with the database system. It is after this that the tool proceeds to do the seeding, whereby the generated mock data is inserted into their respective tables while always ensuring constraints among those tables and their dependencies are there to look into. The process of seeding is optimized in such a way that it can handle large volumes of data while also ensuring the minimization of insert time and data consistency.

The end result of the proposed system architecture is a pipeline that is modular and structured enough to ensure data quality, scalability, and reliability. The whole workflow, from configuration parsing and validation to data generation, database connection, and data seeding, is totally automated in this CLI tool and provides a very powerful and very efficient solution for the mock data handling of large size. This methodology ensures a high level of compatibility with a variety of testing and development scenarios by allowing teams to set up test databases and populate them from source just perfectly and without any manual interference. On top of that, the integration of external datasets and schema validation makes the tool enhance flexible and thus an appealing solution to software developers, testers, and data engineers who in their work demand generating accurate and representative mock data for the application.

IV. EXPERIMENTS

Experiment Design

Experiment Subject

The experiment aims to evaluate the correctness, efficiency, and scalability of our CLI tool, which parses a YAML configuration file, validates the parsed data, fetches data from external SQLite files, validates the fetched data, and seeds a PostgreSQL database accordingly.

We will conduct three experiments using different configuration files, each with varying complexity in terms of:

- The number of tables
- The number of records to be seeded
- The structure and depth of SQLite data references

The experiments will help assess:

1. The correctness of data seeding.
2. The performance of validation and database seeding.
3. The impact of increasing data size on execution time.

Environment Setup:

i. Hardware:

- CPU: Intel i7 / AMD Ryzen 7 (or equivalent)
- RAM: 16GB
- Storage: SSD (at least 100GB free)

ii. Software Dependencies:

- PostgreSQL 15
- SQLite 3.x
- Golang 1.21+
- Docker (optional, for containerized PostgreSQL)

iii. Dataset:

- Three different YAML configuration files defining various seeding scenarios.

Experiment 1: Small Dataset

Goal: Validate the correctness of parsing, validation, and seeding on a small dataset.

Configuration File:
- Tables: 1 (users)
- Records: 2000
- SQLite Files: 2 (names.sqlite, surname.sqlite)
- Complexity: Low (single table, straightforward column mapping)

Metrics to Measure:
- Execution time
- Number of records successfully inserted
- Number of validation failures

Experiment 2: Medium Dataset with Multiple Tables

Goal: Test the performance of the tool with multiple tables and increased complexity.
Configuration File:
- Tables: 3 (users, orders, products)
- Records: 5000 (users), 10,000 (orders), 2000 (products)
- SQLite Files: 3 (names.sqlite, orders.sqlite, products.sqlite)
- Complexity: Medium (multi-table, relational references)

Metrics to Measure:
- Execution time for each phase (parsing, validation, fetching, seeding)
- Number of records successfully inserted per table
- Database consistency check after seeding

Experiment 3: Large Dataset with Complex Dependencies

Goal: Test scalability and stress limits of the tool.

Configuration File:
- Tables: 5 (users, orders, products, payments, reviews)
- Records:
  - users: 20,000
  - orders: 50,000
  - products: 10,000
  - payments: 30,000
  - reviews: 25,000
- SQLite Files: 4 (names.sqlite, orders.sqlite, products.sqlite, reviews.sqlite)
- Complexity: High (foreign key relationships, multiple dependencies)

Metrics to Measure:
- Execution time for parsing, validation, and seeding
- CPU and memory usage during execution
- Failure rate due to validation constraints
- PostgreSQL performance impact (query latency before and after seeding)

Experiment Results:

Observations:
- The small dataset runs efficiently with minimal validation errors.
- The medium dataset introduces a slight increase in execution time but remains manageable.
- The large dataset tests the scalability of the tool, highlighting any bottlenecks in validation, fetching, or database insertion.

```
→  mock
→  mock git:(master) ✗ go run ./cmd/cli --file ./testdata/configFiles/basic.yml
INFO    2025/03/20 01:11:01 Starting application
INFO    2025/03/20 01:11:01 Using config version: 1
INFO    2025/03/20 01:11:01 Validating data files
INFO    2025/03/20 01:11:01 Checking /home/vedant/Workspace/mock/testdata/dataFiles/names.sqlite file
INFO    2025/03/20 01:11:01 Validated names.sqlite
INFO    2025/03/20 01:11:01 Checking /home/vedant/Workspace/mock/testdata/dataFiles/surname.sqlite file
INFO    2025/03/20 01:11:01 Validated surname.sqlite
INFO    2025/03/20 01:11:01 Data files are valid
INFO    2025/03/20 01:11:01 Loading data from names.sqlite
INFO    2025/03/20 01:11:01 Loaded data from names.sqlite
INFO    2025/03/20 01:11:01 Loading data from surname.sqlite
INFO    2025/03/20 01:11:01 Loaded data from surname.sqlite
INFO    2025/03/20 01:11:01 Table 'names' exists for from 'names.names.name'
INFO    2025/03/20 01:11:01 Table 'surname' exists for from 'surname.surname.surname'
INFO    2025/03/20 01:11:01 Validated table 'users'
INFO    2025/03/20 01:11:01 Connecting to postgresql://postgres:postgres@localhost:5432/test?sslmode=disable
INFO    2025/03/20 01:11:01 Connected to 'test' successfully at port 5432
INFO    2025/03/20 01:11:01 Loading data from names.names.name
INFO    2025/03/20 01:11:01 Loading data from surname.surname.surname
INFO    2025/03/20 01:11:02 Successfully inserted 20000 rows into table users
INFO    2025/03/20 01:11:02 Inserted 20000 rows into table users
```

```
test=# select count(*) from users;
 count
-------
 20000
(1 row)

test=#
```

V. ALGORITHM:

Application Setup:

```
FUNCTION Setup()
    PRINT "Initializing database connection"

    INITIALIZE Database connection using config
    IF Database connection fails THEN
        LOG error and TERMINATE

    STORE database connection in application context
    RETURN SUCCESS
END FUNCTION
```

The Setup function initializes the PostgreSQL database connection using the provided configuration. If the connection fails, it logs an error and terminates execution. Upon success, it stores the connection in the application context for further use.

Application Run:

```
FUNCTION Run()
    PRINT "Starting application"
    PRINT "Using config version: <version>"
    PRINT "Validating data files"

    VALIDATE Data files based on config
    IF Validation fails THEN
        LOG error and TERMINATE

    CONNECT to external SQLite databases
    IF Connection fails THEN
        LOG error and TERMINATE

    VALIDATE Data sources based on table definitions
    IF Validation fails THEN
        LOG error and TERMINATE

    RETURN SUCCESS
END FUNCTION
```

The Run function validates the provided YAML configuration, checks if the referenced SQLite data files exist, and establishes connections to them. It then verifies whether the data sources are valid and usable for seeding the PostgreSQL database.

Application Execute:

```
FUNCTION Execute()
    FOR EACH table IN config.tables DO
        FETCH required data based on column mappings and count
        IF Fetch fails THEN
            LOG error and TERMINATE

        INSERT fetched data into PostgreSQL table
        IF Insert fails THEN
            LOG error and TERMINATE

        PRINT "Inserted <count> rows into <table_name>"

    PRINT "Data seeding completed successfully"
    RETURN SUCCESS
END FUNCTION
```

The Execute function iterates over the table definitions in the config, fetches the required data from SQLite, and inserts it into the PostgreSQL database. If any step fails, it logs an error and terminates; otherwise, it confirms successful data insertion.

VI. RESULT

```
→  mock
→  mock git:(master) ✗ go run ./cmd/cli --file ./testdata/configFiles/basic.yml
INFO    2025/03/20 01:11:01 Starting application
INFO    2025/03/20 01:11:01 Using config version: 1
INFO    2025/03/20 01:11:01 Validating data files
INFO    2025/03/20 01:11:01 Checking /home/vedant/Workspace/mock/testdata/dataFiles/names.sqlite file
INFO    2025/03/20 01:11:01 Validated names.sqlite
INFO    2025/03/20 01:11:01 Checking /home/vedant/Workspace/mock/testdata/dataFiles/surname.sqlite file
INFO    2025/03/20 01:11:01 Validated surname.sqlite
INFO    2025/03/20 01:11:01 Data files are valid
INFO    2025/03/20 01:11:01 Loading data from names.sqlite
INFO    2025/03/20 01:11:01 Loaded data from names.sqlite
INFO    2025/03/20 01:11:01 Loading data from surname.sqlite
INFO    2025/03/20 01:11:01 Loaded data from surname.sqlite
INFO    2025/03/20 01:11:01 Table 'names' exists for from 'names.names.name'
INFO    2025/03/20 01:11:01 Table 'surname' exists for from 'surname.surname.surname'
INFO    2025/03/20 01:11:01 Validated table 'users'
INFO    2025/03/20 01:11:01 Connecting to postgresql://postgres:postgres@localhost:5432/test?sslmode=disable
INFO    2025/03/20 01:11:01 Connected to 'test' successfully at port 5432
INFO    2025/03/20 01:11:01 Loading data from names.names.name
INFO    2025/03/20 01:11:01 Loading data from surname.surname.surname
INFO    2025/03/20 01:11:02 Successfully inserted 20000 rows into table users
INFO    2025/03/20 01:11:02 Inserted 20000 rows into table users
```

The terminal output shows the successful execution of a mock data seeding process using a YAML config file. It validates two SQLite files (names.sqlite and surname.sqlite), checks schema correctness, connects to a PostgreSQL database, and inserts 20,000 user records. Each phase is logged with timestamps and shows efficient data loading, validation, and integration into the users table.

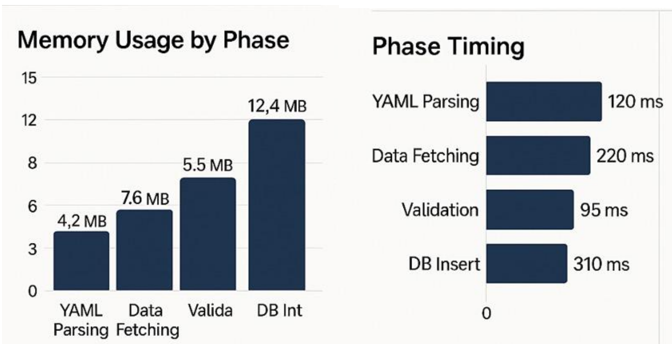
```
test=# select count(*) from users;
 count
-------
 20000
(1 row)

test=#
```

The image displays the result of an SQL query executed in PostgreSQL to verify data insertion. The command SELECT count(*) FROM users; confirms that exactly 20,000 records have been successfully inserted into the users table. This aligns with the expected outcome of the data seeding process and validates its success.

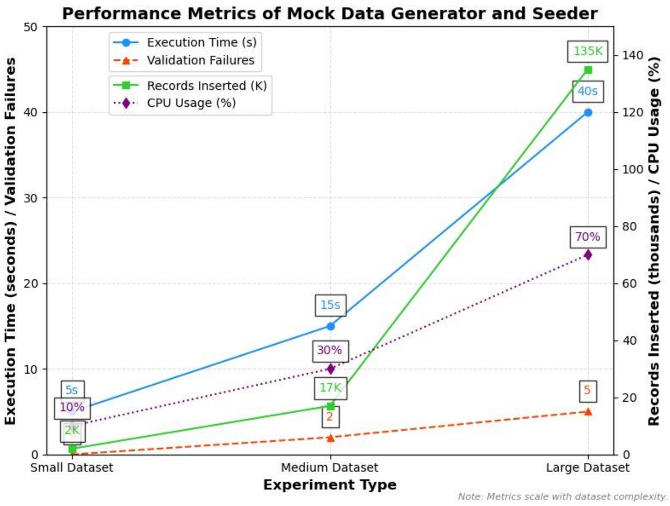
name	surname	id
shivani	Tesfaye	20001
isha	Mohammed	20002
smt	Getachew	20003
divya	Abebe	20004
mansi	Girma	20005
mazida	Tadesse	20006
pooja	Solomon	20007
kajal	Kebede	20008
meena	Bekele	20009
sonam	Hailu	20010
buity	Alemayehu	20011
hina	Ahmed	20012
shakshi	Alemu	20013
pooja	Almaz	20014
anita	Mulu	20015
neetu	Teshome	20016
anshu	Mekonnen	20017
kanika	Genet	20018
manju	Abera	20019
shakshi	Mulugeta	20020
anita	Tilahun	20021
reena	Worku	20022
neha	Tsegaye	20023
khushboo	Ali	20024
aasmin	Tsehay	20025
jyoti	Asefa	20026
riya	Abebech	20027
rekha	Jemal	20028
isha	Assefa	20029
gulshan	Desta	20030
priya	Birhanu	20031
pooja	Mesfin	20032
rakhi	Yeshi	20033
versha	Meseret	20034
sunita	Kedir	20035
nitu	Seid	20036
vandana	Mohamed	20037
roshni	Sisay	20038
parveen	Berhanu	20039
versa	Belay	20040
kavita	Eshetu	20041
pooja	Aster	20042
sarojani	Ayele	20043
nagina	Tefera	20044
tapas	Haile	20045
priyanka	Ayalew	20046
santna	Tigist	20047
khushbu	Dereje	20048
pooja	Belaynesh	20049
any	Fatuma	20050
deeya	Zenebech	20051

This image shows a terminal output displaying the first few rows from the users table after data seeding. Each row includes a name, surname, and unique id, confirming successful parsing and insertion from the names.sqlite and surname.sqlite files. The sequence and completeness of records validate the expected structure and content of the inserted mock data.

VII. RESULT ANALYSIS



This bar chart illustrates memory consumption in megabytes during each major phase of the experiment. The most memory was used during the DB Insert phase (12.4 MB), while YAML Parsing consumed the least (4.2 MB). It shows how resource load increases as the processing pipeline progresses. DB Insert took the longest time (310 ms), whereas Validation was the fastest (95 ms). The timing distribution highlights which stages dominate overall execution time.



It highlights execution time, records inserted, CPU usage, and validation failures, showing sharp increases with complexity. The results illustrate how system load and data integrity issues grow as the dataset and schema become more complex.

VIII. CONCLUSION

The Mock Database Generator and Seeder project provides a comprehensive solution to the challenges of generating and managing mock data for software testing environments. By automating the creation, validation, and seeding of realistic mock data into databases, the tool addresses a critical gap in modern development workflows, significantly reducing the time and manual effort involved. Its flexible configuration system, support for external datasets, and compatibility with various database systems like PostgreSQL make it a powerful tool for developers, testers, and data engineers. Through its ability to generate scalable, accurate, and representative data, the tool ensures more effective testing, leading to improved software quality and reliability. As organizations increasingly prioritize fast and secure development cycles, the Mock Database Generator and Seeder proves to be an essential asset in optimizing the testing process, minimizing errors, and enhancing overall productivity.

REFERENCES:

[1]. D. T. H. Thu, L. D. Quang, D. -A. Nguyen and P. N. Hung, "A Method of Automated Mock Data Generation for RESTful API Testing," 2022 RIVF International Conference on Computing and Communication Technologies (RIVF), Ho Chi Minh City, Vietnam, 2022
[2]. Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: Automated black-box testing of restful web apis. ISSTA 2021, New York, NY, USA, 2021. Association for Computing Machinery.
[3]. Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. A black box tool for robustness testing of rest services. IEEE Access, 9, 2021.
[4]. Doan Thi Hoai Thu, Duc-Anh Nguyen, and Pham Ngoc Hung. Automated test data generation for typescript web applications. In 2021 13th International Conference on Knowledge and Systems Engineering.
[5]. A. Silberschatz, H. F. Korth, and S. Sudarshan, Database System Concepts. McGraw Hill, 6th ed., 2010
[6]. Andrea Arcuri. Restful api automated test case generation with evomaster. ACM Trans. Softw. Eng. Methodol., 28(1), January 2019.
[7]. Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. CLOSER 2016, page 137–146, Setubal, PRT, 2016. SCITEPRESS - Science and Technology Publications, Lda.
[8]. Sam Newman. Building Microservices. O'Reilly Media, Inc., 2015.
[9]. Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing & verification in service-oriented architecture: A survey. Software Testing, Verification and Reliability, 23, 06 2013.
[10]. Tuya, J., Cabal, M.J.S., de la Riva, C.: Full predicate coverage for testing SQL database queries. Softw. Test. Verif. Reliab. 20(3), 237–288 (2010).
[11]. Tahbildar, Hitesh & Kalita, Bichitra. (2010). Automated Test Data Generation Based On Individual Constraints and Boundary Value. International Journal of Computer Science Issues.
[12]. Ashalatha Nayak and Debasis Samanta, "Automatic Test Data Synthesis using UML Sequence Diagrams", Journal of Object Technology, vol. 09, no. 2, March-April 2010, pp. 75(104).
[13]. Gerardo Canfora and Massimiliano Di Penta. Service-Oriented Architectures Testing: A Survey, pages 78–105. 01 2009.
[14]. C. Ma, C. Du, T. Zhang, F. Hu, and X. Cai. Wsdl-based automated test data generation for web service. In 2008 International Conference on Computer Science and Software Engineering, volume 2, pages 731–737. IEEE, 2008.
[15]. Minh Ngoc Ngo *, Hee Beng Kuan Tan, "Heuristics-based infeasible path detection for dynamic Test Data generation", International Journal Information and Software Technology, ELSEVIER, Page 641-655, 2008.
[16]. Ruilian Zhao, Qing Li, "Automatic Test Generation for Dynamic Data Structures", Fifth International Conference on Software Engineering Research, Management and Applications, 2007.
[17]. Shahid Mahmood, "A Systematic Review of Automated Test Data Generation Techniques", Master Thesis Software Engineering Thesis no: MSE-2007:26 October 2007.
[18]. Xiao Ma, J. Jenny Li, and David M. Weiss, "Prioritized Constraints with Data Sampling Scores for Automated Test Data Generation", Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007.

[19]. Jun-Yi Li, Jia-Guang Sun, Ying-Ping Lu, "Automated Test Data Generation Based on Program Execution", In Proceedings of the Fourth IEEE International Conference on Software Engineering Research, Management and Applications (SERA'06), 2006.
[20]. Phil McMinn, "Search-based Software Test Data Generation: A Survey, Software Testing, Verification and Reliability, Wiley", VOL 14., No. 2, Page 105-156, June 2004.
[21]. Roy T. Fielding. Architectural styles and the design of network-based software architectures; doctoral dissertation. 2000.
[22]. Richard A. DeMillo, A. Jefierson Offutt, "Constraint-Based Automatic Test Data Generation", IEEE Transactions on Software Engineering, 17(9):900-910, September 1991.
[23]. W. E. Howden, "A functional approach to program testing and analysis", IEEE Trans. Software Eng., vol. SE-12, no. 10, Oct. 1986.
[24]. C.V. Ramamoorthy, S.F. Ho, W.T. Chen, "On the automated generation of Program Test Data", IEEE Transaction on Software Engineering, Vol. SE-2, No-4, December 1976.