Journal of Simulation

ISSN: 1747-7778 (Print) 1747-7786 (Online) Journal homepage: www.tandfonline.com/journals/tjsm20

Model input verification of large scale simulations

Rumyana Neykova & Derek Groen

To cite this article: Rumyana Neykova & Derek Groen (19 May 2025): Model input verification of large scale simulations, Journal of Simulation, DOI: 10.1080/17477778.2025.2490133

To link to this article: https://doi.org/10.1080/17477778.2025.2490133

JOURNAL OF SIMULATION
https://doi.org/10.1080/17477778.2025.2490133

REVIEW ARTICLE

OPEN ACCESS

Check for updates

Model input verification of large scale simulations

Rumyana Neykova and Derek Groen

Department of Computer Science, Brunel University London, London, UK

ABSTRACT
Reliable simulations require accurate input data. Invalid values, missing data, and format inconsistencies can cause crashes or result distortions, compromising the findings. This paper presents a methodology for verifying the validity of input data in simulations, a process we term model input verification (MIV). We implement this approach in FabGuard, a toolset that uses established data schema and validation tools for simulation modelling. We formalize MIV patterns and create a verification pipeline for existing workflows. FabGuard's applicability is demonstrated across three domains: conflict-driven migration, disaster evacuation, and disease spread models. We also explore Large Language Models (LLMs) for automating constraint generation. In a migration simulation case study, LLMs correctly inferred 22/23 developer-defined constraints, identified errors in existing constraints, and proposed new, valid ones. Our evaluation demonstrates that MIV is feasible on large datasets, with FabGuard processing 300 input files in 140 seconds and maintaining consistent performance across file sizes.

ARTICLE HISTORY
Received 22 August 2024
Accepted 24 March 2025

KEYWORDS
Simulations; verification; validation; schema inference and generation; input data verification

1. Introduction

Simulations have become an indispensable tool across various scientific disciplines, offering insights into complex systems ranging from epidemiology and environmental science to social dynamics and engineering in many different ways (Epstein, 2008). Recent advancements in computational power and data analytics have enabled researchers to develop and apply more realistic and actionable simulation approaches that deliver benefits in a growing number of areas. For instance, epidemiological simulations have been used to inform public health interventions during the COVID-19 pandemic on the national level (Ferguson et al., 2020), as well as hospital-level allocation decisions on the local level (Mahmood et al., 2022). Furthermore, in environmental science, simulations have provided insights into ecosystem interactions and biodiversity under changing climate conditions (Dada & Mendes, 2011; Geary et al., 2020; Jahani et al., 2023).

These simulations increasingly operate at large scales, characterized by extensive computational requirements, and in agent-based models, millions of interacting agents. For example, epidemiological models on the city level or larger may track billions of individual interactions (Epstein, 2008), while migration simulations like Flee (Suleimenova et al., 2017) can involve hundreds of thousands of agents moving across multiple countries, and require thousands of input files with location data and movement patterns when used to account for different conflict and intervention scenarios. These simulations increasingly operate at large scales, characterized by extensive computational requirements, and in agent-based models, millions of interacting agents. For example, epidemiological models on the city level or larger may track billions of individual interactions (Epstein, 2008), while migration simulations like Flee (Suleimenova et al., 2017) can involve hundreds of thousands of agents moving across multiple countries, and require thousands of input files with location data and movement patterns when used to account for different conflict and intervention scenarios.

Particularly when simulation results inform critical decision-making processes, their reliability and reproducibility becomes of paramount importance (Coveney et al., 2016). Here, the open-source software movement has played a crucial role in promoting software sustainability and reproducibility, particularly in scientific simulations (Coveney et al., 2021). Free and Open Source Software (FOSS) helps to facilitate reliable simulation, because open source models can be freely scrutinized by the wider community. In addition, it stimulates software sustainability in general because external maintainers and contributors deliver a public benefit when contributing to a FOSS project (Coveney et al., 2021). Initiatives such as the Journal of Open Source Software (JOSS) (Smith et al., 2018) and the increasing number of journals requiring code availability demonstrate the

CONTACT Rumyana Neykova rumyana.neykova@brunel.ac.uk Department of Computer Science, Brunel University London, Wilfred Brown Building, Uxbridge, London UB8 3PH, UK

scientific community's recognition of the critical role that software plays in research reproducibility. In the context of simulation, open-source practices not only facilitate peer review of the underlying code but also enable researchers to verify and extend existing models, fostering compounded scientific progress (Benureau & Rougier, 2018). Significant challenges remain, however, as a main barrier to reproducible research is that many of the tools required for reproducibility, such as version control, unit testing, and automation, are often seen as being of interest only to professional coders (Alhozaimy et al., 2017). This perception gap highlights the need for solutions that can make these essential practices more accessible, easy to use and relevant to domain experts who may not have extensive software engineering backgrounds.

While verification, validation and uncertainty quantification have received clear attention from researchers in recent years (see e.g., Coveney et al. (2021)), a crucial and often overlooked aspect of ensuring simulation reliability and reproducibility is the process of validating and verifying model input data. In particular, few generic approaches exist that verify that model input data adheres to predefined constraints that ensure correct simulation execution and that it correctly represents the real-world scenarios being modelled. We call this process Model Input Verification (MIV). This step is essential in guarding against simulation results being corrupted by human data input errors or poorly formatted raw input, and helps to prevent cascading errors or crashes that can arise from such flawed or misrepresented inputs. The implications of inadequate input verification in simulations can be severe. For instance, in 1999 a mistaken unit type in one of the ground software sub-models led to the NASA Mars Climate Orbiter having an incorrect trajectory and burning up in the Martian atmosphere (Stephenson et al., 1999). Similarly, in Flee migration simulations it occasionally happens that developers retrieve GPS coordinates for locations in their simulation, and accidentally insert the coordinates of identically named places that reside in an entirely different country.

Recent years have seen a growing emphasis on testing data and ensuring data quality, forming the basis for test-driven data analysis and "unit tests" for data (Schelter, Lange, et al., 2018). Libraries such as Pandera,[1] Great Expectations,[2] and Cerberus[3] have emerged to verify data constraints and validate schemas. These tools have proven valuable in fields such as data science and business intelligence, where they help maintain data integrity and detect errors early in the analysis pipeline (Bantilan, 2020).

However, simulation development often occurs in environments quite different from traditional software engineering. Typically, these simulations are created by domain experts—scientists, researchers, and analysts—who, while highly skilled in their fields, may not have extensive programming backgrounds (Merali, 2010). This gap between domain expertise and software engineering practices has long been a challenge in ensuring the reliability and verifiability of scientific simulations (Roy & Oberkampf, 2011; Thacker et al., 2004). Moreover, the tools and approaches for data validation have not been widely translated to simulations, are often unavailable to simulation practitioners, and the simulation inputs often require constraints that go beyond simple data validation. For example, in agent-based models of population displacement, input verification must ensure not only that population values are non-negative but also that the sum of populations across different locations matches the total simulated population. Furthermore, temporal consistency in the input data is crucial; in disease spread models, the order and timing of intervention measures must be aligned with the simulation timeline.

To address the aforementioned challenges and improve the reliability of simulations, this paper presents FabGuard,[4] an integrated set of tools and methods for Model Input Verification. Our work is guided by several key research questions: How can we effectively adapt existing data validation tools to the unique needs of simulation modelling? What are the types of input verification constraints that a model should support? How can we incorporate input verification into existing simulation workflows? How does the performance and scalability of MIV tools hold up when processing large-scale simulation datasets with varying complexities? To what extent can Large Language Models help domain experts adopt MIV by assisting with constraint generation? How does the performance and scalability of MIV tools hold up when processing large-scale simulation datasets with varying complexities? To what extent can Large Language Models help domain experts adopt MIV by assisting with constraint generation?

In addressing these questions, our work offers several novel contributions to the field.

§ 3.1 Introduces Fabguard, a streamlined verification pipeline that can be easily integrated into CI/CD workflows of simulation models, promoting automated input verification.

§ 3.2 Formalizes model input verification requirements for simulation modelling. We present a framework categorizing constraint types across various dimensions of simulation input data, offering a systematic approach to address verification needs.

§ 4 Demonstrates the practical applicability of FabGuard across three diverse simulation domains: conflict-driven migration, disaster evacuation, and disease spread models. This showcases the adaptability of off-the-shelf tools for input verification in complex simulation scenarios.

§ 5 Presents the first exploration of LLMs Presents the first exploration of LLMs for constraint generation and inference in the context of Model Input Verification, investigating their potential to assist simulation practitioners with initial adoption.

§ 6 Evaluates FabGuard's performance providing insights into its scalability and efficiency in various scenarios.

Section 2 discusses related work, and Section 7 concludes with a summary of contributions and future directions.

2. Related work

The problem of reproducibility in computational science has been identified as a critical issue (Coveney et al., 2016), and there are ongoing efforts to address it (Coveney et al., 2021). Automated testing is needed to systematically verify computer simulations, a precondition to ensuring that the results they produce are sufficiently robust to inform decision-making in the real world (Coveney & Highfield, 2021). This section contextualizes the role of input verification within the broader domain of simulation modelling and further explores solutions in the fields of data analytics and data workflows, which face similar challenges.

Before discussing specific approaches, we clarify key terms as they are used in this paper, following established ASME[5] definitions. Verification is "the process of determining that a computational model accurately represents the underlying mathematical model and its solution". Validation refers to "determining the degree to which a model accurately represents the real world". Uncertainty quantification involves "identifying, quantifying, and assessing the impact of uncertainty sources in simulation". While these processes form an interconnected framework for simulation reliability, our work focuses on input verification as a foundational step.

Verification of simulations is crucial for ensuring that computational models accurately represent real-world scenarios and for enhancing reproducibility. Various approaches and tools have been developed to enhance this process. Code verification focuses on identifying programming errors and verifying numerical algorithms through Software Quality Assurance (SQA) procedures, ensuring software reliability and consistency (Thacker et al., 2004). Comprehensive frameworks for Verification, Validation, and Uncertainty Quantification (VVUQ) further improve predictive capabilities by incorporating methods to estimate and propagate uncertainties through models (Roy & Oberkampf, 2011). Several frameworks and large toolkits have been developed to address these challenges. For example, the VECMA toolkit (Groen et al., 2021) offers a suite of tools for verification, validation, sensitivity analysis, and uncertainty quantification. Within VECMAtk, EasyVVUQ (Wright et al., 2020) streamlines VVUQ for computationally expensive simulations and extensive sampling spaces. FabSim3 (Groen et al., 2023), a Python-based automation toolkit, reduces human effort in simulation-based research and provides an auto-validation tool for comparing simulation accuracy. The Model Verification Tools (MVT) framework (Russo et al., 2022) offers mechanisms for VVUQ assessment of agent-based models, including sensitivity analysis techniques. Uncertainpy (Tennøe et al., 2018) facilitates robust simulation modelling by offering uncertainty quantification and sensitivity analysis using quasi-Monte Carlo and polynomial chaos expansions methods. For a comprehensive overview of many works on verification, validation and especially for uncertainty quantification, readers are directed to (Coveney et al., 2021).

Beyond these specific tools, there are more general works addressing various aspects of simulation verification and validation. Gundersen (Gundersen, 2021) emphasize the importance of transparency and openness as key drivers for reproducibility (Roungas et al., 2018). address the challenge of selecting appropriate V&V methods due to the abundance of available techniques, proposing a methodology for choosing the optimal methods based on simulation characteristics. In the realm of high-performance computing, Encinas et al. (Encinas et al., 2019) present a simulation model of HPC I/O systems using Agent-Based Modelling and Simulation (ABMS), providing insights into I/O performance behaviour in different configurations. Farrell et al. (Farrell et al., 2011) highlight the importance of automated continuous testing in numerical modelling, demonstrating significant improvements in code quality and programmer efficiency (Sinisi et al., 2021). address interoperability challenges in Cyber-Physical System (CPS) simulation, presenting an implementation of FMI 2.0 functions for improving efficiency in simulation-based V&V. These diverse approaches collectively contribute to ongoing efforts to improve the reliability, efficiency, and reproducibility of simulation-based research across various domains.

Despite these advancements, there remains a notable gap in addressing model input verification. Most existing tools and frameworks focus on verifying simulation code, quantifying uncertainties, or validating outputs, rather than verifying input data. The current paper addresses this crucial aspect of simulation reliability by focusing specifically on model input verification, thus complementing existing VVUQ approaches.

Data validation and verification have gained significant attention in the data science and machine learning communities. Schelter et al. (Schelter, Lange, et al., 2018) introduced the concept of "unit tests" for data, providing a framework for describing data constraints. This has led to research on data schema generation, inference, and validation techniques for complex machine learning applications (Hynes et al., 2017; Pimentel et al., 2017; Schelter, Böse, et al., 2018). Modern machine learning platforms now incorporate explicit data validation components, addressing issues such as data drift, model performance degradation, and input data quality (Jha, 2019; Patel et al., 2023; Shankar et al., 2023; Siddiqi et al., 2023; Smith et al., 2018; Wong et al., 2023).

The growing emphasis on data quality and schema verification has led to the development of several tools and libraries aimed at streamlining these processes. Great Expectations (Great Expectations Team, 2024) has emerged as a popular tool for data validation and documentation, allowing users to express their data expectations in a declarative manner and facilitating automated testing of data quality. Pandera (Bantilan, 2020) provides a flexible and expressive API for performing data validation on pandas DataFrames, enabling the definition of schemas with column-level and dataframe-level validation rules, including complex statistical checks. Other tools like Cerberus (Iarocci, 2024) offer similar functionality, reflecting a broader trend towards more robust, automated approaches to data validation across various domains. The TDDA Python module[6] supports test-driven data analysis through various tools, including Reference Testing for managing complex data analysis pipeline tests and tools for discovering, validating and detecting anomalies in data constraints.

These developments in data validation techniques and tools provide a strong foundation for addressing similar challenges in the simulation domain. While the focus of these works has primarily been on data science and machine learning applications, many of these approaches and tools can be adapted or repurposed for simulation input verification. In the context of the extensive literature on VVUQ for simulations, input verification is acknowledged but still not deeply explored. However, as simulations become more complex and reproducibility becomes a more pressing concern in scientific research, the role of input verification will become increasingly prominent.

3. MIV overview

This section provides an overview of Model Input Verification, its importance in simulation modelling, and introduces FabGuard as a comprehensive toolset for implementing MIV. We begin by explaining the concept and significance of MIV. We then present a formalism for categorizing different types of input verification tasks, which serves as a framework for understanding and implementing MIV processes. Finally, we introduce FabGuard, detailing its architecture and key features.

Model Input Verification is an important step in the simulation modelling process, ensuring that input data adhere to specified constraints and accurately represent the real-world scenarios being modelled. In essence, MIV allows users to write tests that check whether input files meet specific requirements and satisfy a set of predefined constraints. These tests help prevent cascading errors that can arise from flawed or misrepresented inputs, enhancing the reliability and reproducibility of simulation results. Common MIV tasks include checking data types, value ranges, inter-column relationships, and cross-file consistency.

To illustrate our approach and the main ideas behind model input verification, we use as a running example an agent-based simulation, called Flee (Suleimenova et al., 2017). Flee is designed for modelling displacement and migration patterns, enables researchers to create simulations based on conflict and disaster scenarios and helps to predict how populations move in response to various crises. It has been applied in major research initiatives such as the EU-funded HiDALGO[7] and ITFLOWS[8] projects. the EU-funded HiDALGO[9] and ITFLOWS[10] projects. Within Flee, agents move across a location graph defined by two primary input CSV files: locations.csv, which defines the nodes of the graph representing various locations such as towns, camps, and conflict zones, and routes.csv, which defines the edges of the graph, representing possible paths between locations.

3.1. MIV workflow

In Figure 1 we present the high-level methodology for writing MIV tests. Here the first two stages, selecting input files and identifying dependencies, are manual processes performed by the user. These manual steps are important for establishing the context and scope of the verification process, while FabGuard is designed to support and automate the subsequent stages, providing a plugin-based architecture that accommodates various input file formats and validation methods.

3.1.1. Selection of input files

In this initial stage the user selects input files to verify. The format and content will vary and are simulation-

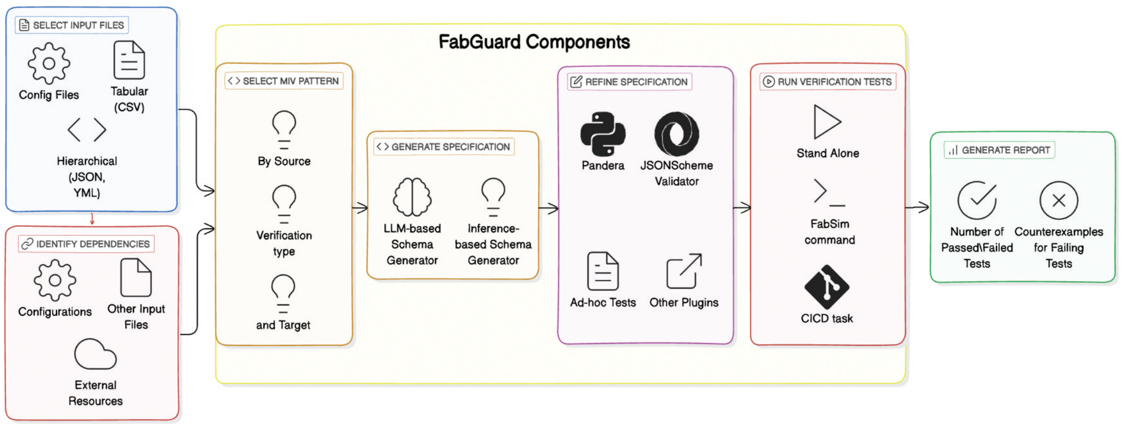


Figure 1. Methodology for model input files verification.

specific. The files are categorized into configuration files, which provide necessary settings for running simulations, and input files that supply the data required to execute processes. For instance, in the Flee simulation tool, the input files might include locations.csv and routes.csv which contain tabular data, while the configuration file is simsettings.yml and contains key-value pairs of simulation parameters.

3.1.2. Identifying dependencies

In the next stage, the user must identify dependencies that are essential for parameterizing the inputs. This involves configurations that require specific settings, supplementary input files that provide context, and external resources such as databases or APIs needed for validation. For example, if simsetting.yml sets the simulation to start on January 1 2023, any closure events in closures.csv with earlier dates should be flagged as invalid.

3.1.3. Selecting MIV patterns

After identifying input files and their dependencies, users should determine which MIV patterns (as formalized in Section 3.2) are appropriate for their verification needs. For example, when verifying a single column in a tabular file, pattern MIV 1.A.i might be suitable. For checks involving configuration files, patterns like MIV 4.C.ii would be more appropriate. This pattern selection guides the type of specifications to be generated and helps ensure coverage of verification requirements. For instance, in Flee, verifying population values in locations.csv would use pattern MIV 1. A.i, while checking route consistency between locations.csv and routes.csv would require pattern MIV 2. A.ii. The chosen patterns inform both the verification approach and the tools needed for implementation.
After identifying input files and their dependencies, users should determine which MIV patterns (as formalized in Section 3.2) are appropriate for their verification needs. For example, when verifying a single column in a tabular file, pattern MIV 1.A.i might be suitable. For checks involving configuration files, patterns like MIV 4.C.ii would be more appropriate. This pattern selection guides the type of specifications to be generated and helps ensure coverage of verification requirements. For instance, in Flee, verifying population values in locations.csv would use pattern MIV 1. A.i, while checking route consistency between locations.csv and routes.csv would require pattern MIV 2. A.ii. The chosen patterns inform both the verification approach and the tools needed for implementation.

3.1.4. Generating specifications

Once the user has identified the input files for verification and their potential dependencies, they can begin writing input verification tests. FabGuard supports two off-the-shelf libraries for schema validation (which is a type of verification and should not be confused with the validation in a simulation context), depending on the type and format of the data—Pandera and jsonschema. The former is a library for defining schemas and validating pandas DataFrames; which allows users to define column-level and dataframe-level validation rules, including data types, value ranges, and custom checks. The latter is a lightweight way to test your YAML/JSON files based on how they conform to a defined schema. FabGuard provides a thin wrapper over both Pandera and jsonschema libraries, enabling integration with simulation tools, LLMs, and providing consistent documentation. Users can start writing tests using the library that best suits their case.

However, writing these tests can be a tedious process that requires programming skills, potentially hindering the tool's applicability. To address this, we have explored two potential ways to bootstrap this stage:

(1) Schema Generators: FabGuard supports built-in schema generators—a custom YAML schema generator, and a Pandera inference module. These tools can automatically infer basic constraints such as data types, minimum

and maximum values for most files. While not comprehensive, they create useful scaffolding that can later be refined by users. For instance, a schema generator might infer that the "population" column in locations.csv should contain non-negative integers.

(2) Large Language Models (LLMs): As reported in Section 5, we have explored the use of LLMs for constraint generation and inference. Our findings indicate that LLMs can not only create the scaffolding of the main tests but also suggest and infer novel constraints. For example, an LLM might suggest that the sum of populations across all locations should match the total simulation population, a constraint that might not be immediately obvious to users.

These automated approaches serve as a starting point, providing a basic scaffolding which can then be refined and expanded by domain experts. This stage significantly lowers the barrier to entry for using FabGuard, making it more accessible to researchers who may not have extensive programming experience.

3.1.5. Refining specifications

The test should be further refined, and most importantly, themselves verified. This stage is important, especially if automated inference tools were used in the previous steps. As outlined in Section 5, some constraints, although they can be inferred, may require adjustments to accurately reflect the simulation's requirements. For example, in Flee, an inferred constraint might correctly identify that the "population" field should be non-negative, but may need refinement to specify that conflict zones must have a non-zero population while other location types can have zero population.

The aforementioned previous stages of automated inference are optional, as developers could write all tests from scratch, tailoring them precisely to their simulation's needs. Alternatively, they could write custom checks for specific validation scenarios not covered by standard tools or inferred constraints. For instance, in Flee, a custom check might be needed to ensure that all routes listed in routes.csv correspond to actual connections between nodes specified in locations.csv, a relationship that may not be captured by automated inference tools.

3.1.6. Running tests

FabGuard input verification tests can be run in several ways, as it is integrated with FabSim3 (Groen et al., 2023), a Python-based automation toolkit for scientific simulation and data processing workflows. This integration allows users to run FabGuard tests as part of simulation workflows within FabSim3 or execute them independently for focused input verification. Furthermore, FabGuard tests can be incorporated into Continuous Integration/Continuous Deployment (CI/CD) pipelines, such as GitHub Actions, enabling ongoing automated validation.

3.1.7. Report generation

In the final stage, FabGuard generates a report detailing test results, including the number of passed and failed tests. Counterexamples for failing tests are provided, highlighting where and why certain tests failed and providing insights to guide corrective actions. If locations.csv fails validation due to missing entries, the report pinpoints these omissions, as well as the the exact rows and values which do not satisfy the constraints.

3.2. MIV conceptual overview

The MIV workflow described above encompasses a wide range of verification tasks, each with its own characteristics and requirements. To systematically address these diverse needs, we have developed a formalism that categorizes MIV tasks based on their sources, templates, and targets. This formalism not only provides a common language for discussing MIV tasks but also helps to identify patterns and best practices across different simulation domains.

In this formalism, we define MIV as the act of synthesizing data from one or more different *Sources* to dynamically generate a verification *Template*, which defines the content pattern required to pass verification. This verification Template is in turn applied to an input file (the *Target*) to perform the actual verification, returning a correct outcome if a match is achieved, and an error if not. Now the MIV task can be performed in different ways, and we provide a simple formalism in Figure 2 to help understand the different patterns that can be created.

Here, each pattern is described with a dot-delimited code, consisting of three components: the Source (or sources) using an Arabic numeral symbol, the Template Type using a capitalized letter symbol and the Target using a Roman numeral symbol. We provide two example pattern definitions in Figure 2. For instance, a MIV 1.A.i pattern could be a check that all locations in a geographic location file have a population of at least 0, while a pattern of type MIV 4.C.ii might (i) check whether the simulation is configured to explicitly model flooding events and then (ii) check whether locations in that same geographic file have, for example, an altitude and water holding capacity value defined if this is the case.

Sources that may be used to generate the template may be content from the target file itself (1, as in our MIV 1.A.i example), from other input files (2), external reference information such as a lookup table or calendar (3) or simulation configuration files (4, as in

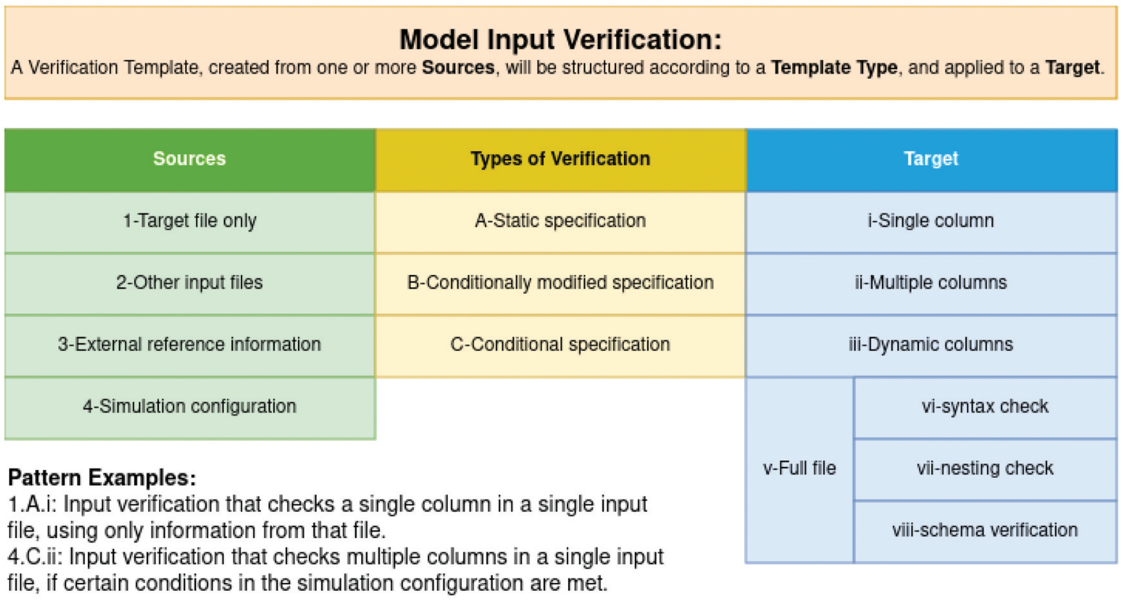

Model Input Verification:
A Verification Template, created from one or more Sources, will be structured according to a Template Type, and applied to a Target.
Sources
Types of Verification
Target
1-Target file only
A-Static specification
i-Single column
2-Other input files
B-Conditionally modified specification
ii-Multiple columns
3-External reference information
C-Conditional specification
iii-Dynamic columns
4-Simulation configuration
v-Full file
vi-syntax check
vii-nesting check
viii-schema verification
Pattern Examples:
1.A.i: Input verification that checks a single column in a single input file, using only information from that file.
4.C.ii: Input verification that checks multiple columns in a single input file, if certain conditions in the simulation configuration are met.

Figure 2. Overview of the MIV formalism.

our MIV 4.C.ii example). It can be possible that a MIV pattern draws from multiple sources, such as the target file (1) and simulation configuration files (4). In this case the Arabic numerals can be appended in numerical order, giving the value "'14'" for the first component in this case. MIV can be of different types, because they can be applied in different ways. These types include specifications that are statically applied to check a file (A, as in our MIV 1.A.i example), specifications that may be modified depending on specific criteria (B), specifications that may or may not be applied depending on specific criteria (C, as in our MIV 4.C.ii example), or (BC) specifications that may be modified or not be applied depending on specific criteria. Normally, MIV of type A tends to be done either using only the target file as source (1.A.*) or the simulation configuration (4.A.*).

Lastly, MIV patterns may differ in which aspect of the input file they verify, i.e., what they target. For instance, they may target an individual column in a tabular data file (i, as in our MIV 1.A.i example), multiple static columns in a tabular data file (ii, as in our MIV 4.C.ii example), a dynamic number or arrangement of columns in a tabular data file (iii). There are also MIV patterns that target files as a whole, and may target non-tabular model input files (v and higher). This may be done specifically to verify the syntax of the input file (vi), the nesting structure (vii, particularly useful for YAML-based input files) and the adherence to a predefined schema (viii, useful for both XML and YAML files for instance).

Given the three components and their variations, we are therefore able to define a total of at least 72 MIV patterns, and more if we include patterns that rely on multiple Source types. However, the range of MIV patterns is not intended to be exhaustive, and there are valuable input verification checks that we chose to leave outside of this formalism to retain simplicity. Most of these verification checks are checks that operate on 0 or multiple files, such as verification checks that operate on network-fed input data, checks that verify the number of input files present or checks that verify the non-existence of redundant or possible disruptive input files.

3.3. *MIV in the context of HPC*

Our MIV tool can be applied to any application that requires input files in one of the supported formats. For large-scale HPC simulations, where millions of agents execute in parallel, it provides essential safeguards against input errors that could waste computational resources and invalidate results across parallel runs. This value is amplified when applications and input files are shared among multiple users.

The SEAVEA project (Software Environment for Actionable VVUQ-enabled Exascale Applications, https://www.seavea-project.org), has established tools where this is the case. The toolkit itself provides facilities for the verification, validation and uncertainty quantification of HPC applications, and is an extension of the VECMA toolkit (Groen et al., 2021). For instance, within FabSim3 (Groen et al., 2023) there are established plugins that contain sample input files for a range of different application domains. There are plugins available for applications in various domains, such as migration, Covid-19, climate, materials and fusion. Here, our tool allows users to verify the input files present in the shared repository, and improve the quality of the input configurations for all other users.

The SEAVEA toolkit, and in particular FabSim3, also provides facilities to simplify the use of the MIV tool. For instance, FabSim3 enables external tools to be used through simple one-liner bash commands, automatically locating the relevant configuration files for the user's application using its internal database. In addition, invocations of the MIV tool can be directly integrated into existing FabSim3 commands. Through this integration, users can choose to apply input file verification automatically for their own daily simulation workflows. Although such automated MIV checking introduces a performance overhead of several seconds, it ensures that any input files that the user requires are verified without additional human effort.

4. Exemplars

This section demonstrates the capabilities of the MIV toolchain by going through common input verification scenarios. To showcase the general nature of our tool, we present three exemplars on: (i) conflict-driven migration, (ii) disaster evacuation and (iii) disease spread.

These exemplars were selected to illustrate a range of input verification challenges commonly encountered in simulation modelling. They progress from basic data type checks to more complex multi-file validations and domain-specific constraints. By presenting these diverse scenarios, we aim to demonstrate FabGuard's capability in handling various types of input data, file formats, and validation requirements. By presenting real-world applications, we demonstrate how the tool integrates into existing simulation workflows. These exemplars serve not only as proof of concept but also as guidance for potential users, illustrating how FabGuard can be adapted to different domains and specific verification needs.

We chose to focus on agent-based models (ABMs) for our exemplars due to their diverse applications across scientific disciplines, complex input requirements, and sensitivity to input errors. ABMs typically involve multiple, interconnected input files describing agent characteristics, environment properties, and simulation parameters, providing an excellent testbed for FabGuard's capabilities. Moreover, ABMs are often developed by researchers from diverse backgrounds, aligning with FabGuard's goal of making input verification more accessible to domain experts. While we focus on ABMs in our examples, FabGuard's is extensible to other simulation types. For instance, molecular dynamics simulations using the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [11] contain input files with specialized key-value style commands that define simulation units, atomic masses, force fields and parameters for molecular interactions. Though these files require a custom parser and validation functions due to their specialized command-line format, the underlying validation requirements align naturally with our MIV patterns. For example, parameters must have valid values and relationships (following similar principles to MIV 1.A.i for single-value validation), and data must be consistent across linked input files (similar to MIV 2.A.ii principles). These examples demonstrates how the MIV methodology presented here is adaptable and applies to different simulation paradigms, and the ad-hoc tests and plugins in Figure 1 reflect exactly this scenario.

The techniques demonstrated here highlight FabGuard's ability to improve input verification across diverse computational modelling and simulation scenarios.

4.1. Exemplar 1: Conflict-driven migration modeling with flee

As already mentioned in § 3, Flee (Suleimenova et al., 2017) is a simulation tool designed to model displacement and migration patterns. Flee simulates hundreds of thousands of agents moving simultaneously across multiple countries Flee simulates hundreds of thousands of agents moving simultaneously across multiple countries. It enables researchers to create simulations based on conflict and disaster scenarios, helping to predict how populations move in response to various crises.

Flee models agents that move across a location graph: here, the location graph is defined using two input CSV files (locations.csv and routes.csv). Errors in the location graph input files not only lead to inaccuracies in the simulation, but can also lead to agents getting stuck in certain locations or to Flee to crash altogether. Another important input file for Flee version 3 (Ghorbani et al., 2024) is simsetting.yml, which is used to configure the set of assumptions used in the simulation. Lastly, there are a range of CSV files that define attributes for the spawned agents, as well as for specific locations and routes.

The code snippet in Listing 1 defines a schema for a pandas DataFrame using the pandera class DataFrameModel. It specifies that the DataFrame should have a "population" column with floating-point numbers greater than 0, which can also be null, and a "location_type" column with string values that must be one of "conflict_zone", "town", or "camp". The Check function is used to enforce these constraints, with Check.greater_than (0) ensuring the "population" values are positive and Check.isin(["conflict_zone", "town", "camp"]) ensuring the "location_type" values are within the specified set. This schema validates the DataFrame's structure and data integrity by checking that the columns match the defined types and conditions.

```python
class LocationsScheme(pa.DataFrameModel):
    location_type: Series[pa.String] = pa.Field(
        isin = ["conflict_zone", "town","camp"])
    population: Series[float] = pa.Field(ge=0,nullable=True,coerce=True)
```

Listing 1: Single-column constraints

We can refine the schema further as to accommodate domain-specific constraints that span multiple columns.

MIV 1.B.ii Multi-column constraint

Locations that are conflict zones require a population value strictly higher than 0 (one needs persons to create conflict-driven displacement):

The provided code snippet in Listing 2 defines a custom validation function for a pandas DataFrame using the pandera library. The @pa.dataframe_check() annotation designates the function population_gt_0 as a custom DataFrame validation check. This function ensures that rows with "location_type" equal to "conflict_zone" do not have a "population" value less than or equal to 0. It creates a boolean mask to identify these invalid rows and raises a ValueError with the indices of any invalid rows found. The function then returns a boolean Series indicating which rows are valid. By using the @pa.dataframe_check() annotation, this custom check is integrated into a pandera schema, allowing it to be used in the validation process to enforce specific data constraints.

4.2. *Exemplar 2: Disaster-driven evacuation modelling with DFlee*

Dflee (Jahani et al., 2023) is a variation of Flee which is configured to model disaster-driven population displacement with tens of thousands of agents responding to flood events with tens of thousands of agents responding to flood events. The simulation tool currently is used for flood-driven migration, but extensions to capture other events (such as storms) are in progress.

Like Flee, DFlee relies on a location graph, but depending on the context the location and route attributed may be radically different. Errors in these input files may result in problems similar to Flee, or in a complete lack of spawned agents in the simulations. DFlee also relies on a simsetting.yml, and a number of fields in there need to be defined correctly for the DFlee to be triggered, while other values need to be lined up in a consistent manner to allow DFlee to work in a manner that matches basic logic (e.g., that people are more likely to flee from flooded areas than unflooded ones). When used for flooding, DFlee also requires a flood_level.csv file, which contains the progression of the flooding at each location during the simulation period. Errors within this file may cause flooding to occur at the wrong times, in the wrong places, or with the wrong intensities.

```python
@pa.dataframe_check()
def population_gt_0(cls, df: pd.DataFrame) -> Series[bool]:
    # Define conditions based on 'location_type' and 'population' columns
    mask = ((df["location_type"] == "conflict_zone") & (df["population"] <= 0))

    # Filter the DataFrame to keep only valid rows
    if mask.any():  # Check if any rows meet the condition
        # Print the rowa that do not meet the condition
        raise ValueError(df.index[mask])
    return ~mask
```

Listing 2: Multi-column constraints

MIV 2.A.ii Constraints spanning multiple files

Within Flee, the countries featured in the model are located in locations.csv, but any border closures are defined in closures.csv. We must ensure closures link to the correct countries. (and for instance do not have typos in the country names)

We can apply the same ideas as above: create a boolean mask that identified the invalid rows and raise an errors if such entries are found. One caveat in comparison to the previous example is that we need to load the locations.csv file. The final constraints is implemented in Listing 3.

MIV 3.A.i Custom-function columns constraints

Validating that a day column has valid rows for all days in a month

The code in Listing 4 defines a custom check function check_day_increment using the pandera library, annotated with @pa.check("Day") to specify that it applies to the "Day" column of a DataFrame. The function validates that the values in the "Day" column are incremental integers within a specified range. It sets a minimum value of 0, a maximum

```
@pa.dataframe_check()
def closure_type_country(cls, df: pd.DataFrame) -> Series[bool]:
    dfl = # Load the content of the "locations" file
    # Get a list of countries from the "locations" file
    loc_countries = dfl["country"].tolist()

    # Define a mask to check if the conditions are met
    mask = ((df["closure_type"] == "country")
            & (~df["name1"].isin(loc_countries)
            & (~df["name2"].isin(loc_countries))))

    # ... raise an errors or return the valid entries
```

Listing 3: Constraints across files

```
@pa.check("Day")
def check_day_increment(cls, series: Series[int]) -> Series[bool]:
    min_value = 0  # define your min value
    max_value = get_sim_period_len()  # define your max value
    step = 1  # define the step increment

    # Check if each value is an increment of ‘step‘ within the range [min_value,
    max_value]
    return ((series - min_value)
```

Listing 4: Stepside checks

value determined by reading the configuration file, and a step increment of 1. The function returns a boolean Series indicating whether each value in the "Day" column meets these conditions: being an increment of 1 from the minimum value, and lying within the inclusive range from the minimum to the maximum value. This ensures that the "Day" column contains valid, sequential day values.

MIV 4.C.iii Dynamic columns constraints

When used for flooding, DFlee also requires a flood level.csv file, which contains the progression of the flooding at each location during the simulation period. Errors within this file may cause flooding to occur at the wrong times, in the wrong places, or with the wrong intensities

Listing 5 demonstrate another pattern which allows for dynamic schema validation where the same constraints should be applied to a varied number of columns. In the schema defined below, the number of columns in the flood levels CSV file is unknown, but all columns except the first specify the same type of information: the intensity of the flood for each day for different flood zones. where the rows are the days, and the columns are the flood zones. To realise these constraints, we have defined a class method with_dynamic_columns within a FloodLevelScheme class that dynamically creates schema constraints for a pandas DataFrame. The method reads configuration values to set maximum permissible values for the "Day" and other flood levels columns. It generates fields with these constraints, and specifying value ranges for all columns. These constraints are added to a dictionary and used to create a new class, ExtendedFloodLevelScheme, which inherits from FloodLevelScheme and includes the dynamically generated attributes.

4.3. Exemplar 3: Disease spread modeling with FACS

FACS (Flu And Coronavirus Simulator) (Mahmood et al., 2022) is a computational modelling tool designed to simulate the spread of influenza and coronaviruses such as COVID-19 in various populations and settings. It simulates millions of agents interacting to model disease spread across metropolitan areas. It simulates millions of agents interacting to model disease spread across metropolitan areas. It allows users to explore the impact of different public health interventions, such as social distancing, vaccination, and lockdown measures, on the spread of these infectious diseases.

```
class FloodLevelScheme
    @classmethod
    def with_dynamic_columns(cls, df: pd.DataFrame):
        flood_zone_max_value = #read from config
        #Create contraints for the day column: value range of 0 to day_max_value

        # Add constraints for all but the first columns
        for column in df.columns[1:]:
            ...
            all_other_fields = pa.Field(...
                in_range={"min_value": 0, "max_value": flood_zone_max_value})
            dynamic_attrs[column] = all_other_fields

        # Create a new dynamic class with columns as defined in dynamic_attrs
        return type('ExtendedFloodLevelScheme', (FloodLevelScheme,), dynamic_attrs)
```

Listing 5: Schema with dynamic columns

To configure individual simulations, FACS relies in a wide range of input files. These include input files to provide geographical information (buildings.csv), demographic information (age-distr.csv and needs.csv), disease information (e.g., disease_covid19.yml and mutations.yml) as well as information on interventions (measures.yml) and vaccination types and strategies (vaccinations.yml). Users commonly edit the measures.yml file to assess the efficacy of new intervention scenarios, and this file is relatively complex in terms of structure. Erroneous entries in measures.yml can have wide-ranging results. For instance, interventions may not trigger at all or they may trigger with the wrong intensity.

MIV 3.A.viii Schema-based summation check

All demographic files (e.g. demographic_age, demographic_gender, etc) for FACS and DFlee contains columns which lists representative fractions of the population. Respectively, the sum of all entries in these columns should add up to 1. (the number required could be modified for different use cases)

Listing 6 implements a DemographicScheme class, which inherits from pa.DataFrameModel in the pandera library, includes a custom validation method all_but_first_column_sum_is_1 marked with the @pa.dataframe_check decorator. This method ensures that the sum of the values in all columns, except the first one, equals 1. It iterates through each column (excluding the first), calculates the sum of its values, and checks if it equals 1. If any column's sum is not equal to 1, it appends the column name and its sum to an errors list. If there are errors, the function would report them; otherwise, it returns True, indicating the DataFrame meets the validation criteria.

MIV 1.A.vii Nested entries yaml validation

In addition to having the correct types, yaml entries should be correctly indented as to preserve the intended meaning. For example, the partial_closure section in the measures.yml allows nested entries, such as for shopping centers, hospitals, etc., enabling detailed specifications for various facilities.

A key insight in our FACS verification journey was that the majority of the FACS yaml verification requirements could be met through off-the-shelf schema validation. Capitalizing on YAML's compatibility as a superset of JSON, we utilized a well-known Python library designed for JSON schema validation. This schema not only specifies the types for each data entry but also outlines the structure, including the hierarchy of entries and the allowance for nested entries.

An excerpt from the jsonschema for the measures.yaml file is given below:

This JSON schema implements the requirements for correctly indented YAML entries with nested structures in the *partial_closure* section. It defines *partial_closure* as an object with specific properties (e.g., "leisure", "school") as numbers between 0 and 1. With "additionalProperties" set to false, it strictly limits entries to these predefined types. This ensures a YAML structure where *partial_closure* is the main section, with only the specified facility types indented beneath it, directly translating the schema's hierarchy into proper YAML indentation and preserving the intended nested relationship.

Through these exemplars, we demonstrate how FabGuard can handle a variety of input verification scenarios, from simple data type checks to complex multi-file validations and domain-specific constraints. This range of examples illustrates the tool's potential to enhance the reliability and reproducibility of simulations across different scientific domains.

An important note to make is the significance of model- data alignment in the process of model-input verification. The conceptual model of a simulation fundamentally shapes its data requirements. Our case studies illustrate this relationship. For example, FACS's epidemiological models require data validation based on disease characteristics, with COVID-19 requiring strict age distribution verification due to its age-dependent outcomes. The importance of this conceptual alignment becomes particularly evident when combining models. For instance, the extension of Flee into DFlee required a careful reconsideration of data validation requirements to reflect the new conceptual model while preserving relevant aspects of the original. This process mirrors the broader challenge in computational science of ensuring that data validation evolves alongside our understanding of the systems

```
class DemographicScheme(pa.DataFrameModel):
    # name: Series[pa.String] = pa.Field(nullable=False, alias='#"name"')

    @pa.dataframe_check
    def all_but_first_column_sum_is_1(cls, df: DataFrame) -> bool:
        # Iterate over the names of all columns except the first one
        errors = []
        for column_name in df.columns[1:]:
            column_sum = df[column_name].sum()
            if column_sum != 1:
                errors.append(f"{column_name},{column_sum}")
        if len(errors) > 0:
           # Report the errors
        return True
```

Listing 6: Schema across Multiple files

```

"partial_closure": {
   "type": "object",
   "properties": {
    "leisure": {"type": "number", "minimum": 0, "maximum": 1},
     "school": {"type": "number", "minimum": 0, "maximum": 1},
     "shopping": {"type": "number", "minimum": 0, "maximum": 1},
     "example": {"type": "number", "minimum": 0, "maximum": 1}
   },
   "additionalProperties": false ...
```

Listing 7: Json Schema

being modelled. Through its flexible constraint systems and configuration-dependent validation rules, FabGuard provides the means to clarify the assumptions of the system and verify data accordingly.

5. LLMs for constraints inference and generation

The adoption of Model Input Verification practices faces challenges due to the complexity of setting up verification frameworks and the need for domain-specific knowledge. To address these usability concerns and lower the barrier to entry for MIV, we explored the potential of Large Language Models (LLMs) in bootstrapping the MIV process bootstrapping the MIV process. LLMs, with their ability to understand and generate human-like text, offer a promising approach to inferring constraints from existing data and generating new constraints based on natural language descriptions. This section investigates two key research questions:

(1) RQ1: Can LLMs be used for constraints inference?
(2) RQ2: Can LLMs be used for constraints generation?

While this exploration is preliminary it demonstrates
While this exploration is preliminary it demonstrates
how we can leverage LLMs to make MIV more accessible to simulation practitioners who may not have extensive programming backgrounds or in-depth knowledge of data validation techniques

5.1. RQ1: Constraints inference

To address RQ1, we conducted an experiment using Claude 3.5 Sonnet,[12] a language model developed by Anthropic[13] and released in 2024. Claude's ability to understand and generate code makes it suitable for our constraint inference experiment. We provided Claude with input files for the Flee simulation, along with explanations of the simulations and instructions on using Pandera for validation.

5.1.1. Methodology

Our approach involved several key steps. First, we supplied Claude with the contents of key input files, including locations.csv, routes.csv, and closures.csv for Flee. We then provided detailed explanations of the simulation, including the purpose of each input file. We introduced Claude to Pandera, explaining its use for DataFrame validation and providing examples of how to create schemas and custom checks. Finally, we asked Claude to infer and generate Pandera schemas and checks based on the provided information.

5.1.2. Findings

Table 1 presents a comparison of key constraints inferred by Claude against our manual tests. We categorized the constraints into four types: simple

Table 1. Comparison of constraints in manual tests vs. LLM-Inferred tests.

Category	Manual Test	LLM-Inferred Test	Status
Single-column	Coordinates within [−180, 180]	Latitude [−90, 90], Longitude [−180, 180]	Improved(Corrected)
	Location type in ["conflict_zone", …, "marker", "idpcamp"]	Location type in ["conflict_zone", "town", …]	Partial (missing "marker" and "idpcamp")
	Route distance > 0	Route distance ≥ 0	Improved(Corrected)
	Forced redirection in [0, 1, 2]	Forced redirection in [0, 1, 2]	Exact match
	Closure type in ["location", "country", "links", "camp", "idpcamp"]	Closure type in ["country", "camp"]	Partial (missing "location", "links", "idpcamp")
Multi-column	Population > 0 for camp, town, conflict; = 0 for markers; ≥ 0 for forwarding hub	Population ≥ 0 for all location types	Requires adjustment (less specific)
	Conflict zones must have a conflict date	Conflict zones must have a conflict date	Exact match
	First country in country column applies to all conflict zones	–	Not inferred
	Location names must be unique	Location names must be unique and non-null	Match (Enhanced)
Multi-file	Closure countries (name1, name2) must be valid countries from locations file	Implemented cross-file check for valid countries in closures	Exact Match
	Location names must exist in routes file (as name1 or name2)	Suggested cross-file check for location names in routes	Exact Match

single-column, refined single-column, multi-column, and multi-file. Simple single-column constraints, which only specify column data types, are omitted from the table. Flee contained 12 such constraints across its three input files. Claude precisely inferred 10 of these and enhanced two date-related single-column constraints (represented as integers) by adding a "greater than 0" restriction. Refined single-column constraints involve validations beyond simple data types, such as ranges or set memberships. Multi-column and multi-file constraints involve relationships between multiple columns or files, respectively.

Our experiment revealed that Claude was capable of inferring a wide range of constraints, including some that were not present in our manual tests. In the analysis of Flee's constraints, 22 out of 23 constraints were correctly inferred, with no wrong inferences. Specifically, 17 constraints were precisely inferred, while one constraint was not inferred at all. Two constraints were corrected from their initial incorrect state, namely the longitude range and route checking. Another two constraints were improved and made stronger than initially proposed. Lastly, one constraint was inferred but was weaker than the actual constraint. This analysis suggests that the inference process aligns closely with the constraints generated by an expert (the second author) working on the tool, though some adjustments were needed to fully capture all aspects of the constraints.

Claude generated several constraints absent from manual tests. For routes.csv, it introduced checks for distinct route endpoints, unique location names, and prevention of duplicate routes. In closure.csv, it validated that end dates should be after the start dates and that the non-null value of a column (name2) depends on another column (closure type). Claude also developed two multi-file constraints: ensuring camp closures reference valid camps from the locations file, and identifying isolated locations. The latter was implemented as:

```
# Constraint: Check for isolated locations (not connected by any route)
connected_locations = set(routes_df['name1']) | set(routes_df['name2'])
isolated_locations = set(locations_df['name']) - connected_locations
if isolated_locations:
    print(f"Warning: The following locations are isolated (not connected by any
    route): {isolated_locations}")
```

This check can reveal potential data errors or geographical inconsistencies in the simulation. These AI-generated constraints demonstrate Claude's ability to infer validation rules addressing data integrity, consistency, and cross-file relationships in the Flee system, potentially identifying errors overlooked in manual testing.

5.1.3. Implications

LLMs can effectively infer a wide range of constraints, potentially accelerating the initial stages of MIV development. They can complement manual tests by identifying additional checks that human developers might overlook. However, the accuracy of LLM-inferred constraints can be improved by providing more detailed configuration information, and data.

5.2. RQ2: Constraints generation

For RQ2, we explored Claude's ability to generate specific constraints when provided with clear descriptions of the constraint.

5.2.1. Methodology

Our approach involved providing Claude with detailed descriptions of constraints, using the same format as in the Exemplars section of this paper. We then asked Claude to implement these constraints using Pandera, specifying that the implementation should include necessary imports and class structures. Finally, we manually reviewed the generated code to assess its correctness and completeness in implementing the described constraints.

5.2.2. Findings

Claude demonstrated a high degree of accuracy in generating constraints based on descriptions. Out of 13 constraint descriptions provided, Claude successfully generated 11 correct implementations. Of the remaining two, both required minor adjustments. Table 2 presents examples of constraint descriptions and Claude's implementations. In both cases, Claude accurately translated the con-

Table 2. Examples of constraint descriptions and generated implementations.

Constraint Description	Generated Implementation
"Route distances must be positive numbers"	distance: Series[Float] = pa.Field(gt=0)
"The sum of all entries in demographic probability columns should add up to 1"	@pa.dataframe_check def probabilities_sum_to_one(cls, df: pd.DataFrame) -> bool: prob_columns = [col for col in df.columns if col != 'category'] return all(df[prob_columns].sum(axis=1).between(0.99, 1.01))

straint descriptions into functional Pandera checks. The generated code not only implements the logical constraints but also follows Pandera's syntax and best practices. However, we observed that for more complex constraints, especially those involving configuration-dependent values or specific simulation logic, Claude's implementations required minor adjustments. For example, in constraints involving maximum flood levels in DFlee, Claude initially used hard-coded values, which we needed to replace with configuration-dependent variables.

5.2.3. *Implications*

LLMs can significantly speed up the initial implementation of MIV constraints, particularly for common validation patterns and clearly described requirements. Our findings suggest that while LLMs may not be ready for fully automated constraint generation, they can reduce initial setup complexity. For instance, in our Flee case study, LLMs correctly inferred basic constraints that could serve as starting templates for domain experts.

5.3. *Balancing Model assumptions and data reality*

While automation can accelerate MIV adoption, an important challenge in data validation is the risk of over-constraining data to match model assumptions rather than adapting models when data consistently challenges these assumptions. Data curation and validation have a rich history in simulation research (Hassan et al., 2010; Macal, 2016; Sinclair et al., 2023), yet a common pitfall remains: the tendency to "clean" data to fit model assumptions instead of using validation failures as signals to revisit these assumptions.

Our experience with Flee illustrates this tension. Initially, constraints required non-zero populations in all conflict zones. However, real-world data revealed that conflict zones could temporarily have zero populations due to complete displacement. Instead of forcing data conformity, these validation failures guided us to refine the model's assumptions.

FabGuard supports this balanced approach through:

- Violation reporting which helps in distinguishing between data quality issues and assumption misalignment
- Configuration-dependent validation allowing constraints to evolve with model refinements
- CI/CD pipeline integration making constraint failures trackable, enabling systematic analysis of whether failures indicate data or assumption issues

The use of LLMs for constraint generation further emphasizes this challenge. While LLMs can infer constraints from existing data and documentation, they may inadvertently codify implicit assumptions that deserve scrutiny. This underscores the importance of human oversight in the constraint development process, ensuring that validation rules reflect well-reasoned model assumptions rather than merely enforcing existing patterns in the data.

5.4. *On the potential use of LLMs in MIV*

Our experiments with Claude on the Flee case study demonstrate that LLMs have significant potential in both inferring and generating constraints for Model Input Verification. They excel at identifying a wide range of constraints and can accurately translate natural language descriptions into functional code. This capability is particularly valuable for domain experts who may have deep subject knowledge but limited programming experience. Rather than needing to learn Pandera's API from scratch, experts can use LLM-generated constraints as templates, modifying them based on their domain understanding. This capability is particularly valuable for domain experts who may have deep subject knowledge but limited programming experience. Rather than needing to learn Pandera's API from scratch, experts can use LLM-generated constraints as templates, modifying them based on their domain understanding.

This capability is especially important in simulation modelling, often developed by domain experts who may lack extensive programming backgrounds. Our preliminary analysis shows that LLMs, when provided with the right setup—including appropriate structure, classes, and examples—can bridge the gap between domain expertise and software engineering practices, at least in the context of input verification. They make the process of writing constraints more accessible and bring the power of formal specification to domain experts who may not have deep programming knowledge.

While LLMs show promise in MIV, their use presents challenges. Our experience revealed a shift from quick constraint generation to time-consuming validation, emphasizing the need for human expertise. Generating constraints with LLMs was quick, taking less than an hour, but validating their accuracy required a several hours of work. LLM-generated constraints, though technically correct, often proved overly conservative, missing potential valid types not present in sample data. This highlights the importance of comprehensive datasets and domain expert involvement when using LLMs for constraint generation. While LLMs can accelerate initial constraint generation, they complement rather than replace human expertise in the MIV process.

6. Evaluation

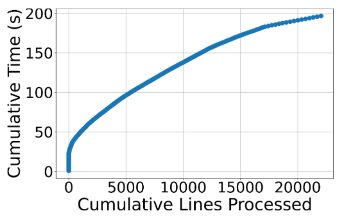
Our evaluation of FabGuard aims to demonstrate its scalability and applicability. We conducted two sets of tests: (1) Microbenchmarks with generated input files and tests; (2) a real-world simulation using the Flee system and custom test files. Section 6.3 presents the microbenchmark results, while Section 6.2 shows the results with FLEE. These tests provide insights into FabGuard's performance across various scenarios, from controlled environments to practical applications. All scripts for generating the benchmarks, are available from the electronic supplementary material.[14]

6.1. Setup

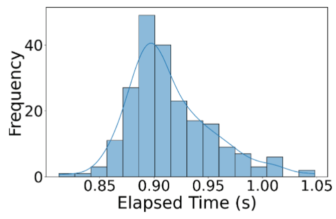
Our evaluation was conducted on an Apple M2 Max with 12–core CPU, 30–core GPU and 16–core Neural Engine, 64 GB of RAM, and 1TB of HDD running MacOS Ventura 13.5. We used Python 3.12.0. To ensure accurate measurements, we employed warm-up runs before collecting performance data. Warmup runs are necessary in benchmarking to allow the system to reach a steady state and minimize the impact of initial system variations (e.g., cache warm-up, background processes). Following established benchmarking methodologies (Georges et al., 2007), we performed 5 warm-up runs to stabilize the JIT compiler, followed by 30 execution runs—a sample size that ensures statistical validity while remaining computationally practical. We report the average execution time across the execution runs. To ensure accurate measurements, we employed warm-up runs before collecting performance data. Warmup runs are necessary in benchmarking to allow the system to reach a steady state and minimize the impact of initial system variations (e.g., cache warm-up, background processes). Following established benchmarking methodologies (Georges et al., 2007), we performed 5 warm-up runs to stabilize the JIT compiler, followed by 30 execution runs—a sample size that ensures statistical validity while remaining computationally practical. We report the average execution time across the execution runs.

6.2. Use case: Flee

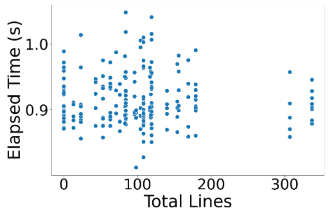
We evaluate FabGuard's performance by running our test suite on the entire Flee conflicts dataset. The test suite consists of three Pandera files implementing all



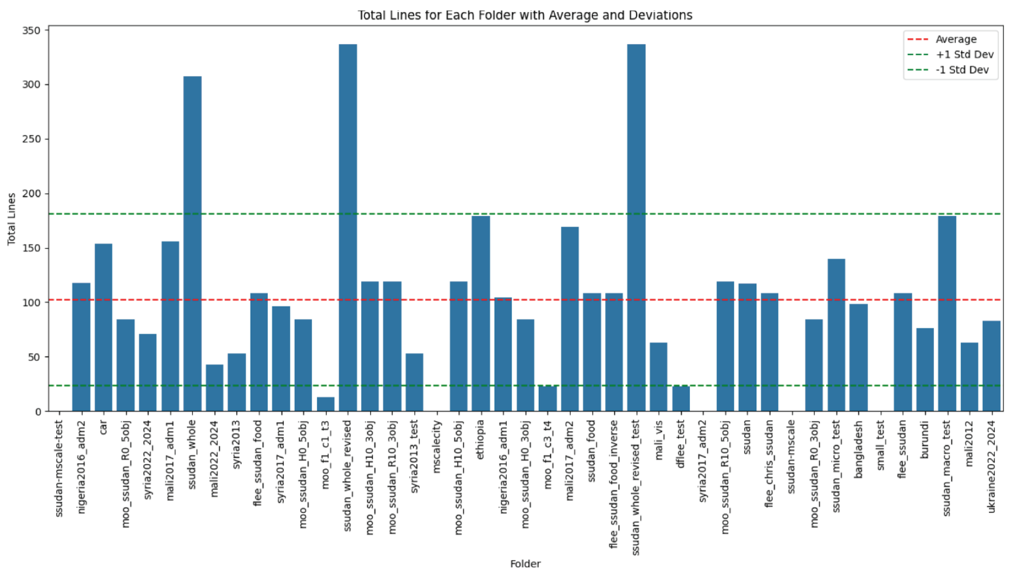
(a) Cumulative lines vs. time



(b) Elapsed time histogram

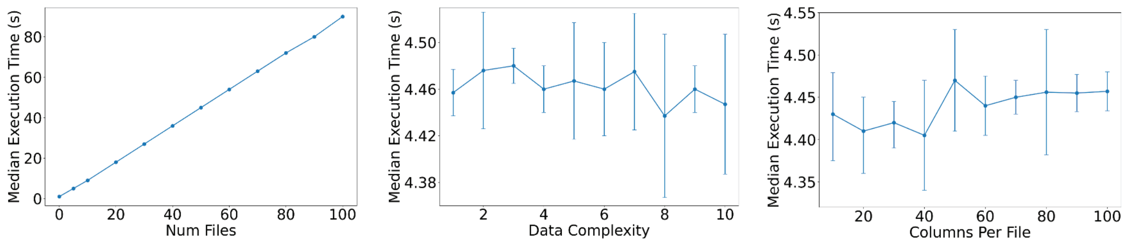


(c) Total lines vs. elapsed time



(d) Total lines per conflict

Figure 3. Analysis of code conflicts and resolution metrics.



(a) Execution time vs. number of files (b) Execution time vs. data complexity (c) Execution time vs. columns per file

Figure 4. Microbenchmark results showing FabGuard's performance characteristics across different dimensions: (a) linear scaling with number of files processed, (b) consistent performance across varying data complexity levels, and (c) stable execution time regardless of the number of columns per file.

23 constraints from Flee, as outlined in Table 1 (column 2 - Manual constraints). Section 4.1 demonstrates representative constraints from each category: single-column constraints (e.g., population values must be non-negative), multi-column constraints (e.g., population requirements for different location types), and multi-file validations (e.g., ensuring closure countries match valid countries from the locations file).

The evaluation covered diverse conflict scenarios displayed in Figure 3(d) covering different conflicts (mali, sudan, syria, etc) along with different scenarios per conflict (e.g., mali_∗). Each conflict (scenario) requires three input files. The locations.csv file defines nodes such as towns, camps, and conflict zones and contains 14 constraints. The routes.csv file specifies connections between locations with 5 constraints, while closures.csv indicates border or route closures with 4 constraints. This dataset allows us to assess FabGuard's efficiency and scalability across a wide range of real-world scenarios, with locations.csv requiring the most complex validation due to its central role in the simulation.

The results of our evaluation are summarized in Figure 3, each subfigure different aspects of FabGuard's performance. Upon analysing these results, several key insights emerge which we have summarised below.

6.2.1. *Scalability*

FabGuard demonstrates good overall scalability, processing approximately 12,000 lines in about 140 seconds (Figure 3(a)).

6.2.2. *Consistency*

The majority of files are processed within a narrow time range of 0.85 to 1.05 seconds, with a peak around 0.90 seconds (Figure 3(b)). This consistency across different file sizes indicates a reliable performance baseline for FabGuard.

6.2.3. *Processing time vs. file size*

Interestingly, there isn't a strong linear relationship between file size and processing time for most files (Figure 3(c)). This suggests that FabGuard has a relatively constant overhead for each file, with the actual content verification time being comparatively small. Moreover, the Flee dataset exhibits significant variability in file sizes across different folders (Figure 3(b)). Most folders contain files with fewer than 200 lines, but some exceed 300 lines. Despite this variability, FabGuard maintains relatively consistent processing times. A few files with longer processing times (1.25–1.27 seconds) create a slight right skew in the distribution (Figure 3(b)), suggesting factors beyond line count can affect processing time.

6.2.4. *Efficiency*

Based on the overall processing of 12,000 lines in 140 seconds, FabGuard achieves an average processing rate of approximately 85.71 lines per second. This rate demonstrates FabGuard's efficiency in handling large datasets. FabGuard ability to process a large number of files quickly makes it suitable for real-world applications where rapid input verification and makes it a viable part of a CI/CD pipeline.

6.3. *Microbenchmarks*

We designed a series of microbenchmarks aimed at stress-testing our approach under various conditions. Taking locations.csv and its corresponding tests as a baseline due to their complexity, we evaluated FabGuard's behaviour across four key dimensions: data types (1–10), number of columns (10–100), rows (100–1000) per file, and total number of files processed (1–100). For the column variation tests, we augmented locations.csv with additional randomly generated columns of different data types (int, float, str, bool, date, etc), creating corresponding simple constraints for validation. For file quantity and data volume tests, we scaled our baseline by generating multiple variations of the file with randomized content while preserving the data structure. These systematically generated tests allowed us to simulate diverse scenarios FabGuard might encounter in real-world applications.

Results, displayed in Figure 4 revealed consistent performance across these input dimensions, with no significant bottlenecks or scalability issues. While slight fluctuations in execution time were observed with changes in data complexity and file structure, these variations were minimal, typically within a range of 0.05 to 0.1 seconds (approximately 1–2% of total execution time). The most notable finding was a linear correlation between the number of files processed and execution time, indicating predictable scaling for large-scale simulations.

7. Discussion and conclusion

Previous research on validation and verification of simulations has established robust frameworks for model validation (Gürcan et al., 2013; Sargent, 2013a, 2013b, 2015), but these primarily focus on verifying model structure and validating outputs rather than systematically verifying input data. While approaches like statistical methods (Cheng, 2006; Law, 2020), independent verification frameworks (Robinson & Brooks, 2010), credibility models (Yilmaz & Liu, 2022), and debugging approaches (R. Gore et al., 2015; R. J. Gore et al., 2017) have advanced simulation validation, they typically address input verification only tangentially. For agent-based simulations specifically, Gürcan et al. (Gürcan et al., 2013) introduced a testing framework with micro, meso, and macro levels of validation, yet still focused primarily on model behaviour rather than verification of input data. In parallel, researchers have explored data gathering and curation methods for agent-based models, with Sinclair et al. (Sinclair et al., 2023) proposing hybrid data gathering approaches for crowd simulations, Bell and Mgbemena (Bell & Mgbemena, 2018) demonstrating data-driven exploration of agent behaviour, and Zhong et al. (Zhong et al., 2022) surveying data-driven crowd modelling techniques. Research on simulation interoperability by Tolk (Tolk, 2024) and composability frameworks by Benali and Ben Saoud (Benali & Ben Saoud, 2011) has further highlighted the importance of conceptual alignment in simulation contexts.

Our Model Input Verification (MIV) framework addresses a gap in simulation validation by offering a methodology focused on input data quality and consistency. Model input verification helps to run simulation correctly, particularly when its configuration is complex. Examples of such complex application scenarios include forecasts that rely on ensemble simulations (Ferguson et al., 2020), simulations consisting of multiple models (Borgdorff et al., 2014) or when a single model is heterogeneously distributed in nature (Groen et al., 2011). While Sargent (Sargent, 2013a) identified data validity as one aspect of verification and data gathering approaches and (Bell & Mgbemena, 2018; Sinclair et al., 2023) have improved approaches for data collection, systematic model input verification has remained underdeveloped. MIV introduces a formalization of verification patterns that can be applied across diverse domains and modelling paradigms, as well as an implementation showcasing their added value. Our approach complements the established model verification (and validation) methods proposed by Sargent (Sargent, 2013a) by providing patterns for verifying input data for such models. That being said, the emphasis of our work is somewhat more applied, which means that the reader benefits from a prototype tool that is shown to work with (and provide added value for) several applications, at the expense of a conceptual framework that might not (yet) capture all possible types of model input verification.

In terms of composability and interoperability, Tolk (Tolk, 2024) in particular notes that conceptual alignment is crucial for meaningful simulation interoperability. In this work, we partially achieved conceptual alignment by proposing application-agnostic model input verification patterns and by implementing our MIV approach as a plugin for the FabSim3 automation toolkit, allowing its application for the full FabSim3 application spectrum (Groen et al., 2023). That being said, the design of our MIV formalism is based on our experience with simulations across disciplines, and it is conceivable that the introduction of a new external application would require the definition of a new MIV pattern (in terms of sources, verification type or verification target).

Overall, our primary contribution is a methodology for MIV, implemented in the FabGuard toolset. This methodology adapts established data schema and validation tools to address the unique challenges of simulation input verification. We formalized MIV patterns, categorizing verification tasks based on their sources, template types, and targets. This formalism provides a structured approach to identifying and implementing input verification requirements across diverse simulation domains.

Our work goes beyond theoretical frameworks by demonstrating the practical application of these MIV patterns. We presented numerous examples across three domains: conflict-driven migration, disaster evacuation, and disease spread modelling. These case studies showcase how FabGuard can handle a variety of validation scenarios, from simple data type checks to complex multi-file validations and domain-specific constraints. Furthermore, we conducted the first study on using Large Language Models (LLMs) for constraint discovery and generation in the context of MIV. Our results show that LLMs can accurately infer existing constraints and even identify new, valid constraints, potentially lowering the barrier to entry for adopting robust MIV practices. This exploration of LLMs, combined with our identified requirements for MIV tools, establishes a foundational framework for the future development of model input verification systems. Our evaluation provided empirical evidence of MIV's feasibility for large-scale simulations, with FabGuard efficiently processing 12,000 lines of data in 140 seconds while maintaining consistent performance across varying file sizes and complexities.

These contributions establish a foundation for more robust and trustworthy simulation practices. We envision MIV becoming an integral part of the simulation modelling workflow, akin to unit testing in software development. Future research will focus on expanding FabGuard's capabilities to cover a broader range of simulation paradigms and input formats. We plan to conduct large-scale studies on the use of Large Language Models, for automated constraint discovery in complex, domain-specific relationships. This research will aim to further lower the barrier for MIV adoption and improve its effectiveness across diverse simulation domains. We will work on developing user-friendly interfaces to make MIV more accessible to non-technical users, bridging the gap between domain expertise and software engineering practices. We will further explore the integration of MIV with other stages of the simulation life cycle, such as output validation and uncertainty quantification. This holistic approach could lead to a more robust framework that can enable more reliable and actionable simulations in a systematic and accessible manner. Furthermore, we will undertake case studies across diverse scientific domains to refine and validate MIV methodologies, providing empirical evidence of their effectiveness and generic application.

This research contributes to establishing input verification as a fundamental component of the simulation modelling process, rather than an afterthought. By integrating MIV into standard modelling practices, we aim to enhance the reliability of simulations and, consequently, the quality of scientific discoveries based on these models. The broader adoption of systematic input verification techniques has the potential to improve the overall robustness and credibility of simulation-based research across various disciplines.

Notes

1. https://www.union.ai/pandera
2. https://greatexpectations.io/
3. https://pypi.org/project/Cerberus/
4. Upon acceptance, the code will be made available on zenodo.
5. https://www.asme.org/codes-standards/publications-information/verification-validation-uncertainty
6. https://github.com/tdda/tdda
7. HiDALGO (https://hidalgo-project.eu/)
8. ITFLOWS (https://www.itflows.eu/)
9. HiDALGO (https://hidalgo-project.eu/)
10. ITFLOWS (https://www.itflows.eu/)
11. https://www.lammps.org/
12. claude.ai.
13. https://www.anthropic.com/
14. The electronic supplementary material will be made available on Zenodo upon paper acceptance.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

The work was supported by the Engineering and Physical Sciences Research Council [EP/W007762/1]; the SEAVEA ExCALIBUR project, which has received funding from EPSRC under grant agreement EP/W00771/1.

References

Alhozaimy, S., Mawdsley, D., Mulholland, D., & Wikfeldt, T. (2017). Towards reproducibility in research software. *Software Sustainability Institute*. Retrieved April 30, 2025, from https://www.software.ac.uk/blog/towards-reproducibility-research-software

Bantilan, N. (2020). Pandera: Statistical data validation of pandas dataframes. In M. Agarwal, C. Calloway, D. Niederhut, & D. Shupe (Eds.), *Proceedings of the 19th Python in Science Conference 2020 (SciPy 2020)* (pp. 116–124). https://doi.org/10.25080/Majora-342d178e-010

Bell, D., & Mgbemena, C. (2018). Data-driven agent-based exploration of customer behavior. *Simulation*, *94*(3), 195–212. https://doi.org/10.1177/0037549717743106

Benali, H., & Ben Saoud, N. B. (2011). Towards a component-based framework for interoperability and composability in modeling and simulation. *Simulation*, *87*(1–2), 133–148. https://doi.org/10.1177/0037549710373910

Benureau, F. C. Y., & Rougier, N. P. (2018). Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. *Frontiers in Neuroinformatics*, *11*, 69. https://doi.org/10.3389/fninf.2017.00069

Borgdorff, J., Mamonski, M., Bosak, B., Kurowski, K., Belgacem, M. B., Chopard, B., Groen, D., Coveney, P. V., & Hoekstra, A. G. (2014). Distributed multiscale computing with muscle 2, the multiscale coupling library and environment. *Journal of Computational Science*, *5*(5), 719–731. https://doi.org/10.1016/j.jocs.2014.04.004

Cheng, R. C. H. (2006). Validating and comparing simulation models using resampling. *Journal of Simulation*, *1*(1), 53–63. https://doi.org/10.1057/palgrave.jos.4250009

Coveney, P. V., Dougherty, E. R., & Highfield, R. R. (2016). Big data need big theory too. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and*

Engineering Sciences, *374*(2080), 20160153. https://doi.org/10.1098/rsta.2016.0153

Coveney, P. V., Groen, D., & Hoekstra, A. G. (2021). Reliability and reproducibility in computational science: Implementing validation, verification and uncertainty quantification *in silico*. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *379*(2197), 20200409. https://doi.org/10.1098/rsta.2020.0409

Coveney, P. V., & Highfield, R. R. (2021). When we can trust computers (and when we can't). *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *379*(2197), 20200067. https://doi.org/10.1098/rsta.2020.0067

Dada, J. O., & Mendes, P. (2011). Multi-scale modelling and simulation in systems biology. *Integrative Biology*, *3*(2), 86–96. https://doi.org/10.1039/c0ib00075b

Encinas, D., Naiouf, M., De Giusti, A., Mendez, S., Rexachs, D., & Luque, E. (2019). On the calibration, verification and validation of an agent-based model of the hpc input/output system. *Proceedings from The Eleventh International Conference on Advances in System Simulation (SIMUL 2019)*, Valencia, Spain.

Epstein, J. M. (2008). Why model? *Journal of Artificial Societies and Social Simulation*, *11*(4), 12.

Farrell, P. E., Piggott, M. D., Gorman, G. J., Ham, D. A., Wilson, C. R., & Bond, T. M. (2011). Automated continuous verification for numerical simulation. *Geoscientific Model Development*, *4*(2), 435–449. https://doi.org/10.5194/gmd-4-435-2011

Ferguson, N. M., Laydon, D., Nedjati-Gilani, G., Imai, N., Ainslie, K., Baguelin, M., Bhatia, S., Boonyasiri, A., Cucunubá, Z., & Cuomo-Dannenburg, G., et al. (2020). *Report, 9: Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand* (Vol. 16). Imperial College London.

Geary, W. L., Bode, M., Doherty, T. S., Fulton, E. A., Nimmo, D. G., Tulloch, A. I. T., Tulloch, V. J. D., & Ritchie, E. G. (2020). A guide to ecosystem models and their environmental applications. *Nature Ecology & Evolution*, *4*(11), 1459–1471. https://doi.org/10.1038/s41559-020-01298-8

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (pp. 57–76). ACM, ACM, New York, NY, USA.

Ghorbani, M., Suleimenova, D., Jahani, A., Saha, A., Xue, Y., Mintram, K., Anagnostou, A., Tas, A., Low, W., Taylor, S. J. E., & Groen, D. (2024). Flee, 3: Flexible agent-based simulation for forced migration. *Journal of Computational Science*, *81*, 102371. https://doi.org/10.1016/j.jocs.2024.102371

Gore, R. J., Lynch, C. J., & Kavak, H. (2017). Applying statistical debugging for enhanced trace validation of agent-based models. *SIMULATION*, *93*(4), 273–284. https://doi.org/10.1177/0037549716659707

Gore, R., Reynolds, P. F., Jr., Kamensky, D., Diallo, S., & Padilla, J. (2015). Statistical debugging for simulations. *ACM Transactions on Modeling and Computer Simulation*, *25*(3), 1–26. https://doi.org/10.1145/2699722

Great Expectations Team. (2024). *Great expectations*. Retrieved April 30, 2025, from https://greatexpectations.io/

Groen, D., Arabnejad, H., Jancauskas, V., Edeling, W. N., Jansson, F., Richardson, R. A., Lakhlili, J., Veen, L., Bosak, B., Kopta, P., Wright, D. W., Monnier, N., Karlshoefer, P., Suleimenova, D., Sinclair, R., Vassaux, M., Nikishova, A., Bieniek, M. . . . Piontek, T. (2021). Vecmatk: A scalable verification, validation and uncertainty quantification toolkit for scientific simulations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *379*(2197), 20200221. https://doi.org/10.1098/rsta.2020.0221

Groen, D., Arabnejad, H., Suleimenova, D., Edeling, W., Raffin, E., Xue, Y., Bronik, K., Monnier, N., & Coveney, P. V. (2023). Fabsim3: An automation toolkit for verified simulations using high performance computing. *Computer Physics Communications*, *283*, 108596. https://doi.org/10.1016/j.cpc.2022.108596

Groen, D., Zwart, S. P., Ishiyama, T., & Makino, J. (2011). High-performance gravitational n-body simulations on a planet-wide-distributed supercomputer. *Computational Science & Discovery*, *4*(1), 015001. https://doi.org/10.1088/1749-4699/4/1/015001

Gundersen, O. E. (2021). The fundamental principles of reproducibility. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *379*(2197), 20200210. https://doi.org/10.1098/rsta.2020.0210

Gürcan, Ö., Dikenelli, O., & Bernon, C. (2013). A generic testing framework for agent-based simulation models. *Journal of Simulation*, *7*(3), 183–201. https://doi.org/10.1057/jos.2012.26

Hassan, S., Pavón, J., Antunes, L., & Gilbert, N. (2010). Injecting data into agent-based simulation. In K. Takadama, C. Cioffi-Revilla, & G. Deffuant (Eds.), *Simulating interacting agents and social phenomena* (pp. 177–191). Springer.

Hynes, N., Sculley, D., & Terry, M. (2017). The data linter: Lightweight automated sanity checking for ml data sets. *NIPS MLSys Workshop*, *1*(5), 10.

Iarocci, N. (2024). Cerberus: Lightweight, extensible data validation library for python. https://pypi.org/project/Cerberus/

Jahani, A., Jess, S., Groen, D., Suleimenova, D., & Xue, Y. (2023). Developing an agent-based simulation model to forecast flood-induced evacuation and internally displaced persons. *International Conference on Computational Science* (pp. 550–563). Springer.

Jha, S. (2019). Data infrastructure for machine learning. *International Journal for Research in Applied Science and Engineering Technology*, *7*(4), 740–742. https://doi.org/10.22214/ijraset.2019.4133

Law, A. M. (2020). Statistical analysis of simulation output data: The practical state of the art. *2020 Winter Simulation Conference (WSC)* (pp. 1117–1127). IEEE, Orlando, FL, USA.

Macal, C. M. (2016). Everything you need to know about agent-based modelling and simulation. *Journal of Simulation*, *10*(2), 144–156. https://doi.org/10.1057/jos.2016.7

Mahmood, I., Arabnejad, H., Suleimenova, D., Sassoon, I., Marshan, A., Serrano-Rico, A., Louvieris, P., Anagnostou, A., Taylor, S. J., Bell, D., & Groen, D. (2022). FACS: A geospatial agent-based simulator for analysing COVID-19 spread and public health measures on local regions. *Journal of Simulation*, *16*(4), 355–373. https://doi.org/10.1080/17477778.2020.1800422

Merali, Z. (2010). Computational science: . . .error. *Nature*, *467*(7317), 775–777. https://doi.org/10.1038/467775a

Patel, H., Guttula, S., Gupta, N., Hans, S., Sharma Mittal, R., & Lokesh, N. (2023). A data-centric AI framework for automating exploratory data analysis and data quality tasks. *Journal of Data and Information Quality*, *15*(4), 1–26. https://doi.org/10.1145/3603709

Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2017). noWorkflow: A tool for collecting, analyzing, and managing provenance from python scripts. *Proceedings of the VLDB Endowment*, *10*(12), 1841–1844. https://doi.org/10.14778/3137765.3137789

Robinson, S., & Brooks, R. J. (2010). Independent verification and validation of an industrial simulation Model. *Simulation*, *86*(7), 405–416. https://doi.org/10.1177/0037549709341582

Roungas, B., Meijer, S. A., & Verbraeck, A. (2018). A framework for optimizing simulation model validation & verification. *International Journal on Advances in Systems and Measurements*, *11*(1), 137–152.

Roy, C. J., & Oberkampf, W. L. (2011). A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing. *Computer Methods in Applied Mechanics and Engineering*, *200*(25–28), 2131–2144. https://doi.org/10.1016/j.cma.2011.03.016

Russo, G., Parasiliti Palumbo, G. A., Pennisi, M., & Pappalardo, F. (2022). Model verification tools: A computational framework for verification assessment of mechanistic agent-based models. *BMC Bioinformatics*, *22*(S14), 626. https://doi.org/10.1186/s12859-022-04684-0

Sargent, R. G. (2013a). Verification and validation of simulation models. *Journal of Simulation*, *7*(1), 12–24. https://doi.org/10.1057/jos.2012.20

Sargent, R. G. (2013b). An introduction to verification and validation of simulation models. *2013 Winter Simulations Conference (WSC)* (pp. 321–327). IEEE, Washington, DC, USA.

Sargent, R. G. (2015). An interval statistical procedure for use in validation of simulation models. *Journal of Simulation*, *9*(3), 232–237. https://doi.org/10.1057/jos.2014.30

Schelter, S., Böse, J.-H., Kirschnick, J., Klein, T., & Seufert, S. (2018). Declarative metadata management: A missing piece in end-to-end machine learning.

Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., & Grafberger, A. (2018). Automating large-scale data quality verification. *Proceedings of the VLDB Endowment*, *11*(12), 1781–1794. https://doi.org/10.14778/3229863.3229867

Shankar, S., Fawaz, L., Gyllstrom, K., & Parameswaran, A. (2023). Automatic and precise data validation for machine learning. *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management* (pp. 2198–2207). ACM, Birmingham United Kingdom.

Siddiqi, S., Kern, R., & Boehm, M. (2023). SAGA: A scalable framework for optimizing data cleaning pipelines for machine learning applications. *Proceedings of the ACM on Management of Data*, *1*(3), 1–26. https://doi.org/10.1145/3617338

Sinclair, J., Suwanwiwat, H., & Lee, I. (2023). A hybrid data gathering and agent based cognitive architecture for realistic crowd simulations. *Journal of Simulation*, *17*(2), 121–148. https://doi.org/10.1080/17477778.2021.1954487

Sinisi, S., Alimguzhin, V., Mancini, T., & Tronci, E. (2021). Reconciling interoperability with efficient verification and validation within open source simulation environments. *Simulation Modelling Practice and Theory*, *109*, 102277. https://doi.org/10.1016/j.simpat.2021.102277

Smith, A. M., Niemeyer, K. E., Katz, D. S., Barba, L. A., Githinji, G., Gymrek, M., Huff, K. D., Madan, C. R., Mayes, A. C., Moerman, K. M., Prins, P., Ram, K., Rokem, A., Teal, T. K., Guimera, R. V., & Vanderplas, J. T. (2018). Journal of open source software (JOSS): Design and first-year review. *PeerJ Computer Science*, *4*, e147. https://doi.org/10.7717/peerj-cs.147

Stephenson, A. G., LaPiana, L. S., Rutledge, P. J., Mulville, D. R., Bauer, F. H., Folta, D., Dukeman, G. A., Sackheim, R., & Norvig, P. (1999). *Mars climate orbiter mishap investigation board phase i report*. November 10, 1999.

Suleimenova, D., Bell, D., & Groen, D. (2017). A generalized simulation development approach for predicting refugee destinations. *Scientific Reports*, *7*(1), 13377. https://doi.org/10.1038/s41598-017-13828-9

Tennøe, S., Halnes, G., & Einevoll, G. T. (2018). Uncertainpy: A python toolbox for uncertainty quantification and sensitivity analysis in computational neuroscience. *Frontiers in Neuroinformatics*, *12*(49). https://doi.org/10.3389/fninf.2018.00049

Thacker, B. H., Doebling, S. W., Hemez, F. M., Anderson, M. C., Pepin, J. E., & Rodriguez, E. A. (2004). Concepts of Model verification and validation. https://inis.iaea.org/records/egfyy-d4t03

Tolk, A. (2024). Conceptual alignment for simulation interoperability: Lessons learned from 30 years of interoperability research. *Simulation*, *100*(7), 709–726. https://doi.org/10.1177/00375497231216471

Wong, S., Barnett, S., Rivera-Villicana, J., Simmons, A., Abdelkader, H., Schneider, J.-G., & Vasa, R. (2023). Mlguard: Defend your machine learning Model! *Proceedings of the 1st International Workshop on Dependability and Trustworthiness of Safety-Critical Systems with Machine Learned Components* (pp. 10–13). ACM, San Francisco CA USA.

Wright, D. W., Richardson, R. A., Edeling, W., Lakhlili, J., Sinclair, R. C., Jancauskas, V., Suleimenova, D., Bosak, B., Kulczewski, M., Piontek, T., Kopta, P., Chirca, I., Arabnejad, H., Luk, O. O., Hoenen, O., Węglarz, J., Crommelin, D., Groen, D., & Coveney, P. V. (2020). Building confidence in simulation: Applications of easyvvuq. *Advanced Theory and Simulations*, *3*(8), 1900246. https://doi.org/10.1002/adts.201900246

Yilmaz, L., & Liu, B. (2022). Model credibility revisited: Concepts and considerations for appropriate trust. *Journal of Simulation*, *16*(3), 312–325. https://doi.org/10.1080/17477778.2020.1821587

Zhong, J., Li, D., Huang, Z., Lu, C., & Cai, W. (2022). Data-driven crowd modeling techniques: A survey. *ACM Transactions on Modeling and Computer Simulation*, *32*(1), 1–33. https://doi.org/10.1145/3481299