PySynthea: A Python-Native Framework for Scalable Synthetic Healthcare Data Generation

Roberto Cruz* David Rey-Blanco†

TietAI

2026-05-21

Abstract

Synthetic healthcare data is increasingly important for research, education, and machine learning development where access to real patient data is limited by privacy and governance constraints. While Synthea provides a widely adopted framework for generating realistic longitudinal electronic health record data, its current implementation presents adoption barriers for many researchers and data scientists due to deployment complexity and limited integration with modern Python-based workflows.

This paper introduces PySynthea, a Python-native reimplementation of Synthea designed to improve accessibility, extensibility, and interoperability within the scientific Python ecosystem. The framework provides modular synthetic patient generation, configurable healthcare simulation pipelines, and support for standard healthcare data formats while integrating naturally with tools such as pandas and machine learning workflows. By reducing operational complexity and aligning synthetic data generation with the dominant data science ecosystem, PySynthea aims to accelerate experimentation and broaden the use of synthetic healthcare data in research and applied AI development.

Keywords: Synthetic Data, Medical AI, Healthcare Simulation, Computational Healthcare, Open Source Software, Clinical Data Simulation

1 Introduction

The development of artificial intelligence and machine learning systems for healthcare has accelerated rapidly over the last decade, with growing evidence that data-driven models can support clinical decision making, operational planning, population health analytics, and biomedical discovery [1–3]. The training, evaluation, and benchmarking of these systems depend critically on access to large, longitudinal, and clinically realistic electronic health record (EHR) data. In practice, however, real EHR data is constrained by privacy regulations, institutional review processes, contractual restrictions, and risks of re-identification, all of which limit the speed at which experimentation can occur and reproducibility can be achieved [4, 5].

*roberto.cruz@tiet.ai

†david.rey@tiet.ai

Synthetic healthcare data has emerged as a complementary resource that can mitigate these barriers without disclosing information about real individuals. High-quality synthetic EHR datasets enable several activities that would otherwise be difficult or impossible at scale, including reproducibility of clinical informatics studies, benchmarking of machine learning models across institutions, education of clinical and informatics trainees, end-to-end testing of healthcare software and integration pipelines, and the safe pre-training and prototyping of machine learning models prior to their deployment on protected health information [6, 7]. As synthetic data has matured, it has become a recognized component of modern healthcare data engineering and a vehicle for reproducible research in clinical artificial intelligence.

Despite well-developed regulatory frameworks, access to real EHR data remains operationally complex. The Health Insurance Portability and Accountability Act (HIPAA) in the United States [8] and the General Data Protection Regulation (GDPR) in the European Union [9] place substantive constraints on how protected health information may be collected, processed, shared, and retained. Institutional approvals, data use agreements, and ethical review processes routinely add months or years to project timelines, and the resulting datasets are typically constrained to a specific cohort, time window, or analytical purpose. Even when de-identified datasets are released, re-identification risks remain non-trivial, especially when records can be linked to auxiliary information [5]. The combination of legal, organizational, and statistical constraints means that real EHR data is often unavailable for many of the early, exploratory stages of machine learning research.

Synthetic data does not eliminate these concerns entirely, but it does change the operational geometry of the problem. A synthetic dataset that is generated from a defined statistical model rather than from real individuals can be shared across institutions, embedded in tutorials, included in software test suites, or used to validate large data pipelines without disclosing protected information. This makes synthetic generation an enabling technology for both education and reproducible research in healthcare informatics.

Although healthcare informatics historically grew up around enterprise Java and C# stacks, the broader landscape of data analytics and machine learning has consolidated around Python. Numerical and tabular computation in Python is supported by NumPy [10] and pandas [11]; deep learning is dominated by PyTorch [12] and TensorFlow [13]; classical machine learning is widely practiced through scikit-learn [14]; large-scale data processing uses Dask [15] and Apache Spark [16] through PySpark; and the day-to-day environment for many researchers is the Jupyter notebook [17]. Foundation models for clinical text and tabular EHR data are also typically distributed in Python-friendly frameworks such as the Hugging Face Transformers library [18, 19]. The result is a strongly Python-first research ecosystem in which most experimentation, model development, and pipeline orchestration is expected to happen.

In this ecosystem, the cost of bridging into a non-Python tool is non-trivial. It introduces additional moving parts in continuous integration, complicates reproducibility, increases the cognitive load on researchers and students, and creates impedance between data generation and downstream model development. For synthetic healthcare data to fully serve the Python-based ML ecosystem, the data generators themselves need to live within it.

This paper introduces PySynthea, a Python-native reimplementation of the Synthea synthetic patient generation framework [20]. PySynthea preserves the conceptual foundations of Synthea — modular, state-machine-based disease modules; longitudinal simulation of patients from birth to death; demographically grounded population sampling; multi-format export including FHIR — while making the generator first-class within the Python ecosystem.

The contributions of this work are as follows. First, we describe a Python-native implementation

that is installable via standard tooling such as `pip` and `uv` [21], requires no JVM, and exposes a clean Python API alongside a `click`-based command-line interface. Second, we describe the architectural decisions that make the system modular and extensible: a state-machine engine that loads the original Synthea Generic Module Framework (GMF) JSON modules unchanged; a separation between simulation engine, world model, and exporters; and a configuration layer that mirrors Synthea's properties model. Third, we describe interoperability features oriented toward the modern Python data ecosystem, including pandas-compatible exports, FHIR R4 bundles, and design patterns for streaming generation. Fourth, we discuss notebook-friendly workflows, parallel and batched generation, and integration points for downstream machine learning pipelines. Finally, we discuss tradeoffs, limitations, and a research agenda that builds on the Python-native foundation toward GPU acceleration, LLM-augmented module authoring, and reinforcement-learning environments grounded in synthetic clinical trajectories.

The remainder of the paper is organized as follows. Section 2 reviews the original Synthea project and its data model. Section 3 examines the limitations of existing synthetic data generation workflows from the perspective of a Python-first user. Section 4 articulates the design goals of PySynthea. Sections 5 and 6 describe the system architecture and the synthetic patient generation pipeline. Section 7 details Python ecosystem integration. Section 8 covers healthcare data standards and export formats. Section 9 discusses performance and scalability. Section 10 presents example workflows and use cases. Section 11 discusses tradeoffs, Section 12 outlines future work, and Section 13 concludes.

2 Background: Synthea

Synthea is an open-source synthetic patient population simulator originally developed by The MITRE Corporation [20, 22]. The project was motivated by the recognition that many healthcare informatics activities — interoperability testing, software validation, education, prototyping, and benchmarking — do not require real patient data, but do require data that is structurally and statistically realistic. Synthea fills this gap by generating fully synthetic patients together with longitudinal clinical histories that mirror the structure of real electronic health records.

A Synthea generation run produces a population of synthetic individuals along with their associated entities. These typically include demographic attributes (such as age, sex, race, ethnicity, and geographic location), encounters (ambulatory visits, inpatient admissions, emergency visits), conditions, medications, procedures, observations and vital signs, immunizations, care plans, allergies, devices, and imaging studies. Each patient has a complete simulated life trajectory from birth (or initialization) to death or to the end of the simulation horizon, and the resulting record includes both timing and clinical content for every modeled event.

At the core of Synthea is the Generic Module Framework (GMF), a state-machine formalism for encoding disease progression and care pathways. Each module is a directed graph of states — including initial states, simple states, delays, guards, encounters, condition onsets and ends, medication orders, procedures, observations, care plan starts and ends, symptoms, and terminal states — connected by direct, conditional, or probabilistically distributed transitions [20, 23]. At each simulation timestep, a patient's active modules are evaluated, transitions are sampled, and clinical events are emitted into the patient's record.

This formalism has several important properties. It separates clinical content from execution semantics, so domain experts can author or modify modules without touching the simulation engine. It is probabilistic, supporting realistic variability across simulated patients. It is composable, since

modules can invoke submodules and reference each other through shared patient attributes. And it is auditable, since the resulting trajectories can be traced through the underlying state graph.

Around this state-machine core, Synthea layers a population model that draws demographics from real census data, a healthcare system model that includes providers and payers, cost data calibrated against publicly available pricing references, and a set of clinical reference resources such as CDC growth charts, immunization schedules, and biometric correlations.

2.1 Interoperability and Standards

A major contributor to Synthea's adoption is its support for healthcare interoperability standards. The Fast Healthcare Interoperability Resources (FHIR) standard [24, 25] provides a modern, RESTful, JSON- or XML-based representation of clinical resources, and is now the dominant standard for exchanging healthcare data. Synthea natively exports synthetic patients and their records as FHIR bundles, in addition to flat CSV exports, CCDA documents, and other formats. Generated resources use standardized coding systems including SNOMED CT [26] and LOINC [27] for diagnoses and observations.

The combination of realistic longitudinal records and standards-conformant export formats has made Synthea a default tool for validating FHIR servers, training SMART-on-FHIR applications [28], and prototyping healthcare apps without exposing real patient information.

2.2 Current Use Cases

Synthea is used today across several distinct communities. In education, it provides a safe and reproducible substrate for teaching clinical informatics, FHIR, and healthcare data science. In machine learning research and prototyping, it enables benchmarking of pipelines and models against datasets that mirror the structural complexity of real EHRs while remaining fully shareable [6, 7, 29]. In healthcare software development, Synthea is widely used for end-to-end testing of EHR integrations, data engineering pipelines, and interoperability gateways. During the COVID-19 pandemic, Synthea was extended to model COVID-19-specific trajectories and to provide synthetic datasets useful for early research before real data could be made widely available [30].

These use cases share a common pattern: they require synthetic data that is rich, longitudinal, and standards-conformant, and they value the absence of regulatory constraints over absolute statistical fidelity. PySynthea is designed to preserve these characteristics while making the tool itself more accessible.

3 Limitations of Existing Synthetic Data Generation Workflows

This section discusses the practical limitations encountered when using existing synthetic data generation tooling — and Synthea in particular — from the perspective of users whose workflows are centered on the Python ecosystem. These limitations should be read as friction relative to modern data science practice rather than as criticism of Synthea itself, whose contributions to the field are substantial.

3.1 Deployment Complexity

The reference implementation of Synthea is written in Java and is typically operated either via a build system (Gradle), a runnable JAR, or a Docker container. While these mechanisms are robust and well established, they introduce real operational friction in environments that are otherwise Python-centric. A researcher who simply wants to obtain 1,000 synthetic patients in a Jupyter notebook is typically required to install a Java Development Kit, clone a separate repository, invoke an external build command, navigate an output directory of CSV or FHIR files, and then re-ingest the resulting files into pandas or another Python tool [20]. Each of these steps adds latency, surface area for configuration errors, and difficulty for users unfamiliar with the JVM toolchain.

For students, applied researchers, and software engineers whose default workflow is `pip install` followed by `import`, this multi-runtime setup is a meaningful adoption barrier. By contrast, a Python-native generator can be installed with a single command, imported as a library, and invoked directly from the same notebook in which downstream analysis lives.

3.2 Limited Integration with Python Workflows

Beyond deployment, a deeper integration cost arises when Synthea-generated data is consumed by a Python-based analytics or ML pipeline. The natural unit of data interchange in this context is the pandas `DataFrame` or, in larger settings, the Dask or PySpark distributed equivalent. Synthea's outputs, however, must first be materialized to disk in CSV, JSON, or FHIR bundle format, then re-parsed and re-shaped before they can be used. This round-trip introduces serialization overhead, fragile parsing logic for nested FHIR structures, and an explicit gap between generation and analysis.

Modern workflows also place a high premium on interactive iteration: regenerating a small cohort with slightly different parameters in a notebook, inspecting it, and iterating again. A non-Python generator that must be invoked through an external process and whose outputs must be parsed back into the host language is inherently misaligned with this pattern.

3.3 Barriers to Rapid Experimentation

Many of the most interesting use cases for synthetic data are inherently iterative. A researcher exploring how an ML model's performance varies with disease prevalence may want to regenerate populations many times with different parameter values. A developer building a custom disease module may want to make a change, regenerate a small cohort, inspect the resulting records, and iterate within seconds. An educator may want to weave generation into a lecture or a hands-on lab without asking students to set up an external runtime.

These workflows favor frameworks in which generation parameters live as plain Python objects, generation can be invoked in a single function call, and outputs are immediately available as in-memory objects. They are less compatible with workflows that require a build system, a separate process, and file-based round-trips.

3.4 Ecosystem Fragmentation

A consequence of these structural mismatches is a degree of fragmentation between the healthcare simulation ecosystem and the broader ML ecosystem. Many of the most important downstream tools — pandas [11], NumPy [10], PyTorch [12], scikit-learn [14], Dask [15], PySpark [16], Airflow [31],

and Jupyter [17] — live in a single connected graph of imports, configuration files, and packaging conventions. A Java-based generator sits outside this graph and must be bridged through file formats and shell invocations, rather than through library calls and shared in-memory data structures.

The cost of this fragmentation falls disproportionately on the users who are best positioned to benefit from synthetic data: applied ML researchers, students, and small healthcare AI teams that lack dedicated infrastructure engineers.

3.5 Extensibility Challenges

Finally, although Synthea's modules are themselves JSON files that are in principle language-agnostic, modifying the simulation engine itself — for example, to add new state types, alter timestep semantics, change how attributes propagate, or instrument the engine for tracing — requires familiarity with the Java codebase, its build system, and its testing infrastructure. For researchers whose primary expertise is in machine learning and clinical data science rather than enterprise Java engineering, this is a steep on-ramp.

A Python-native engine that mirrors the original semantics can lower this barrier substantially. Researchers can extend the simulation engine using the same language they use for analysis, debug it with familiar tools, and contribute improvements through the same workflow they already use for their machine learning code.

4 Design Goals of PySynthea

PySynthea is designed around a small set of explicit goals that follow directly from the limitations described in the previous section. They are summarized in Table 1 and discussed below.

Table 1: Design goals of PySynthea.

Goal	Description
Accessibility	Install with `pip install` or `uv add`; no external runtime; usable from the first line of a notebook.
Interoperability	First-class integration with pandas, NumPy, PyTorch, Dask, and other tools in the Python data ecosystem.
Reproducibility	Deterministic generation conditional on a seed; explicit configuration; ability to reproduce a cohort exactly across runs.
Extensibility	Modular simulation engine with documented extension points for new state types, exporters, and modules.
Scalability	Multi-threaded and process-based generation, streaming export, and design hooks for distributed and GPU-accelerated futures.
Standards Fidelity	Preserve compatibility with the original Synthea GMF JSON modules and export FHIR R4 bundles, CSV, and JSON.
Notebook-First	Designed for interactive use: small cohorts return quickly, generation can be inspected and re-run interactively.

Accessibility is the central goal: removing the JVM dependency and shipping a pure-Python package

eliminates an entire class of installation problems and brings synthetic data generation within reach of every user who is already running Python. *Interoperability* with the dominant data ecosystem ensures that generated data flows directly into downstream analysis and modeling without unnecessary intermediate steps. *Reproducibility* is a non-negotiable requirement for any tool used in scientific research and is supported by deterministic seeding and explicit configuration.

Extensibility recognizes that synthetic data is rarely useful in only its default form: users routinely need new disease modules, custom exporters, or modifications to the underlying engine, and these tasks must be tractable in the user's own language. *Scalability* acknowledges that synthetic data generation is sometimes a small in-notebook task and sometimes a multi-million-patient batch job, and the framework must accommodate both ends of this spectrum. *Standards fidelity* ensures that PySynthea continues to operate within the broader Synthea ecosystem and produces data that is compatible with downstream tools that already expect Synthea-style outputs. Finally, the *notebook-first* orientation reflects the practical reality that the daily working environment for most users in this space is the Jupyter notebook [17].

5 System Architecture

PySynthea is organized into four primary packages within a single installable distribution. The `engine` package implements the simulation engine, including the module loader, state machine semantics, transition logic, condition evaluation, and the top-level generator. The `world` package contains the domain model: persons, health records, demographics, geographic locations, providers, and payers. The `export` package implements exporters for FHIR R4 and other formats. The `helpers` package provides configuration utilities and common functionality. A separate `resources` tree, packaged with the distribution, holds the disease modules, provider and payer data, geographic and demographic data, cost tables, and clinical reference resources that mirror those of upstream Synthea.

The high-level relationship between these components is shown in Figure 1. The `Generator` object is the main entry point. It is configured by a `GeneratorOptions` object and a `Config` object, and it coordinates the demographic sampler, the simulation loop over individual `Person` instances, the module engine, and the exporters. Each `Person` carries a deterministic random generator seeded from the global seed, a dictionary of attributes, a set of active modules with their current states, and a `HealthRecord` that accumulates clinical events.

The simulation engine is built around three central abstractions. A `Module` represents a single state machine, identified by name and composed of a set of named `State` instances and a designated initial state. A `State` is a typed node in the state machine — for example, an `Encounter`, a `ConditionOnset`, a `MedicationOrder`, a `Delay`, or a `Terminal` — that executes side effects on the patient when entered and exposes a transition policy. A `Transition` selects the next state given the current patient context. The set of supported state types in PySynthea mirrors that of upstream Synthea and includes initial, simple, delay, guard, set-attribute, counter, encounter and encounter-end, condition-onset and condition-end, allergy-onset and allergy-end, medication order and end, care-plan start and end, procedure, vital sign, observation, multi-observation, diagnostic report, symptom, death, call-submodule, device and device-end, supply-list, imaging-study, vaccine, and physiology states.

Modules are loaded at startup from JSON files using the existing Synthea GMF schema. This was an explicit architectural decision: rather than re-author hundreds of disease modules in a new format, PySynthea parses and executes the established GMF JSON directly, which preserves more

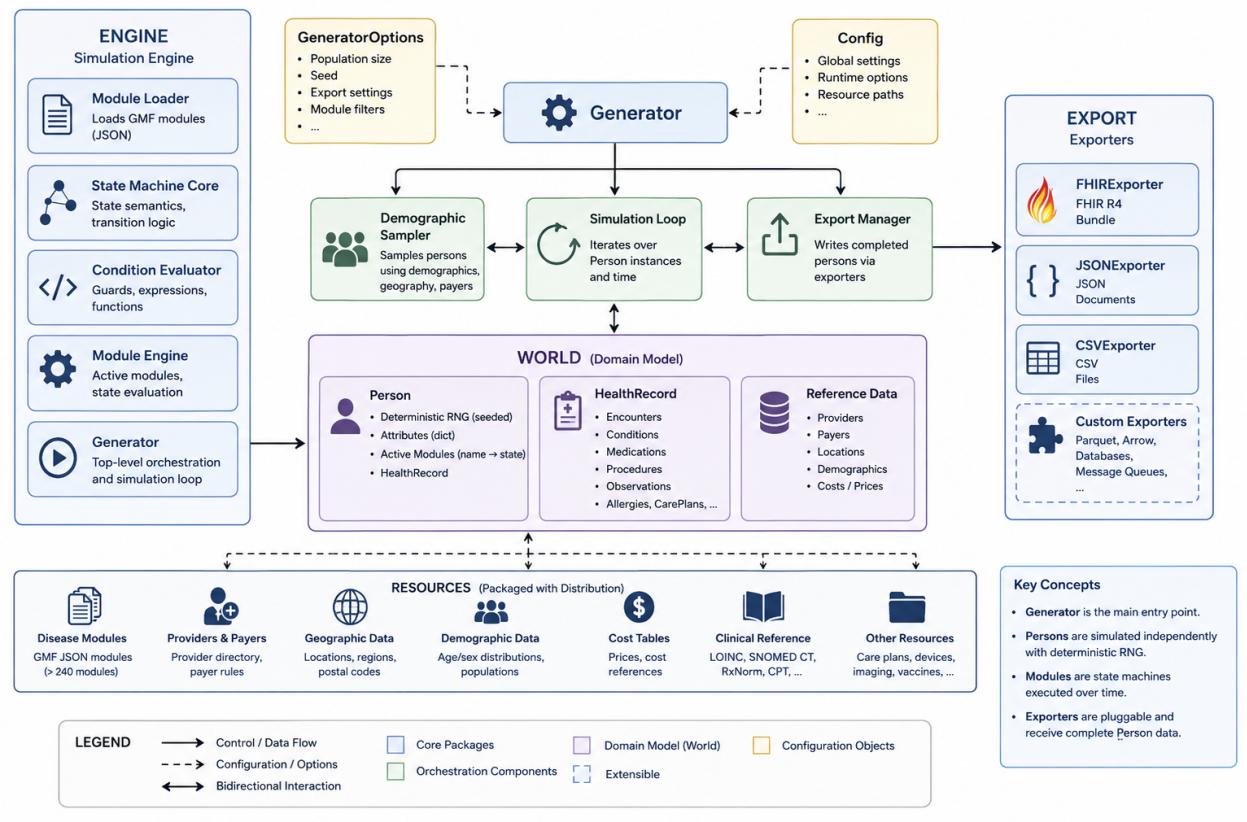

Figure 1: High-level system architecture of PySynthea. The `Generator` orchestrates the simulation loop; `Person` instances carry attributes and a health record; the `Module` engine drives state-machine evaluation over time; and the export layer produces FHIR, JSON, and CSV outputs.

than 240 disease modules and the long history of clinical content embedded in them. From the user's perspective, the same module that runs in upstream Synthea also runs in PySynthea, simplifying validation and enabling incremental contributions to the shared module library.

The exporter layer is structured around a `PatientExporter` interface, with concrete implementations such as `FHIRExporter` producing FHIR R4 bundles. Each exporter receives a fully simulated `Person` and writes its contents to disk or to a downstream sink. The architecture allows new exporters — for example, Parquet, Arrow, or direct database writers — to be added without modifying the simulation engine.

6 Synthetic Patient Generation Pipeline

The synthetic patient generation pipeline in PySynthea proceeds through six conceptual stages. At each stage, the design preserves the semantics of upstream Synthea while exposing Python-native interfaces that are convenient for downstream consumers.

6.1 Population Initialization

When the `Generator` is constructed, it loads the configuration, sets the global random seed, instantiates the demographic sampler over the configured geographic scope, loads providers and payers, and loads the set of active modules. The population size, age range, sex filter, and reference date are taken from the `GeneratorOptions` object. A typical invocation looks as follows.

```
from synthea import Generator, GeneratorOptions

options = GeneratorOptions()
options.population_size = 1000
options.state = "California"
options.city = "San Francisco"
options.seed = 12345
options.threads = 4

generator = Generator(options)
generator.run()
```

6.2 Demographic Sampling

For each requested patient, the demographic sampler produces an initial set of attributes — sex, race, ethnicity, geographic location, ZIP code, and birth date — drawn from real census-derived distributions. The sampler is deterministic when seeded, ensuring that the same set of patients can be regenerated across runs. Social determinants of health, such as income, education, and housing-related variables, can be sampled from accompanying tables [20].

6.3 Disease Progression

Once a patient has been initialized, the engine instantiates the active modules for that patient and begins the main simulation loop. Time advances in discrete steps — by default one week, configurable through the properties file — from the patient's birth date to either their simulated

death or the reference date, whichever comes first. At each timestep, each module is evaluated: the engine inspects the patient's current state in that module, runs the state, samples a transition, and advances the state pointer if the transition completes within the current timestep. Modules can declare delays, which suspend further progression until a specified amount of simulated time has elapsed, and guards, which block progression until a specified condition becomes true.

6.4 Clinical Event Simulation

The states executed during disease progression emit clinical events into the patient's `HealthRecord`. An `Encounter` state opens a new encounter, attaching subsequent condition onsets, medication orders, procedures, and observations until the corresponding `EncounterEnd`. Each event is associated with standardized codes (SNOMED CT for conditions [26], LOINC for observations [27], RxNorm-style codes for medications, and so on) drawn from the module definitions. Cost information is attached to encounters, procedures, medications, immunizations, labs, devices, and supplies using the cost tables shipped with the distribution.

6.5 Temporal Sequencing

A central property of Synthea-style data is its longitudinal nature: events are not isolated, but ordered in time and causally related. PySynthea preserves this through its timestep-based engine. A patient who acquires diabetes early in life may, over the course of decades of simulated time, accumulate medications, laboratory observations, complications, and additional encounters in a clinically plausible order. The same temporal coherence applies across modules — for example, pregnancy modules interacting with prenatal care encounters and immunization schedules.

6.6 Export Generation

Once a patient's life has been simulated, each enabled exporter is invoked with the patient and the simulation reference time. The `FHIRExporter` constructs a FHIR R4 bundle containing the patient resource and each associated event as a separate FHIR resource, while CSV and JSON exporters produce flat tabular and document-oriented outputs respectively. Exporters write to a configured output directory and can be enabled or disabled independently. A schematic of the full pipeline is shown in Figure 2.

7 Python Ecosystem Integration

The defining design choice of PySynthea is to be a first-class member of the Python ecosystem. This section describes the integration patterns that follow from that choice.

7.1 Native DataFrame Access

Because the entire simulation runs in-process, the contents of a patient's `HealthRecord` are immediately available as Python objects without requiring serialization to disk and re-ingestion. PySynthea exposes utilities that materialize these in-memory objects as pandas `DataFrame` instances [11]. A typical pattern in a notebook is to generate a small cohort, immediately materialize the conditions, encounters, observations, and medications as separate DataFrames, and proceed with exploratory

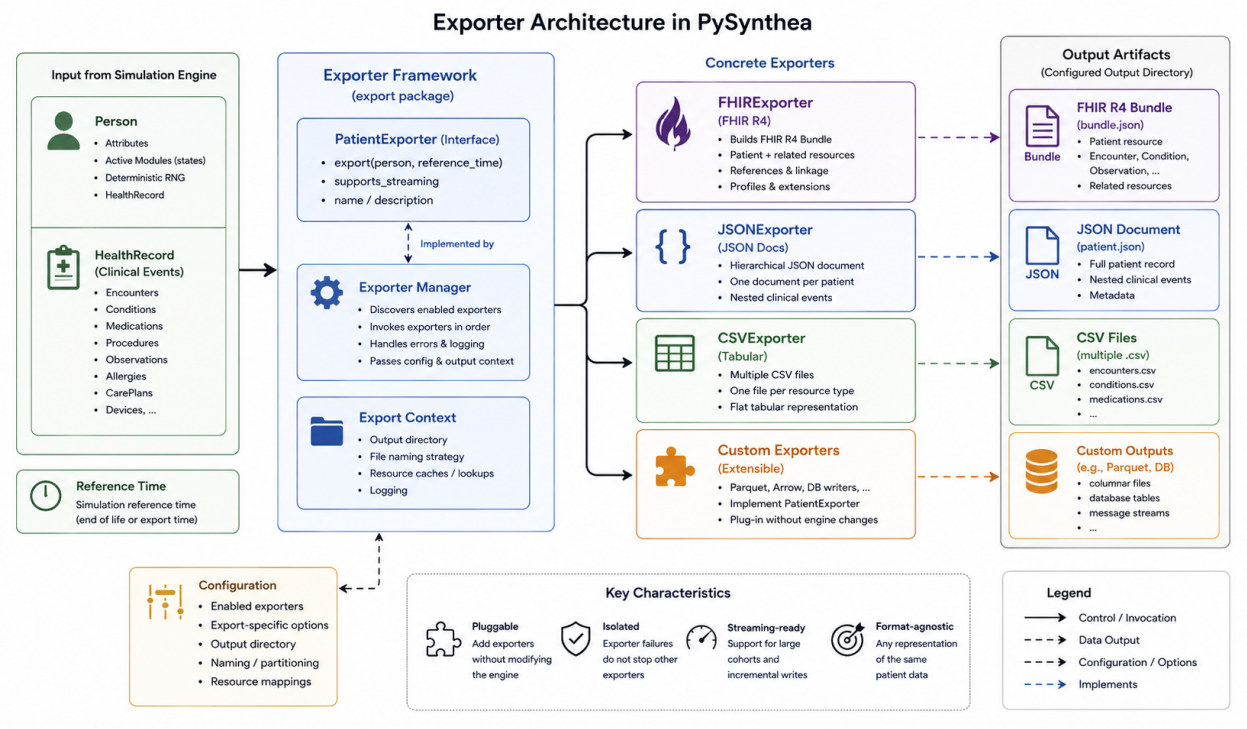

Figure 2: End-to-end synthetic patient generation pipeline. Each patient is initialized, demographically sampled, simulated through the module engine across a longitudinal timeline, and finally exported in one or more formats.

analysis or feature engineering using familiar pandas idioms. This is the same workflow that researchers already use with public benchmark datasets such as MIMIC-III [32], MIMIC-IV [33], and eICU [34], and the synthetic-data path inherits the same ergonomics.

7.2 ML-Ready and Notebook Outputs

For machine learning use cases, in-memory access also makes it natural to map records to tensors. A common requirement is to produce a sequence of clinical events per patient — for example, an ordered list of (timestamp, code, value) triples — suitable for input to a sequence model or transformer [1, 19, 35]. Because the simulation has full control of event timing, the framework can emit such sequences directly. Conversion to PyTorch [12] or TensorFlow [13] tensors is then a simple wrapping step around the already-materialized pandas DataFrames.

PySynthea is designed for the Jupyter notebook environment [17]. Because installation is a single `pip` invocation and the entire API is exposed as ordinary Python classes, a new user can move from `pip install` to a generated cohort in a small handful of cells. The `Generator` is constructed from a `GeneratorOptions` object that is itself a plain Python object, so users can tweak parameters interactively, regenerate cohorts, and inspect the resulting records without leaving the notebook. The CLI, implemented with `click`, mirrors the same API at the shell level and shares its configuration model.

7.3 Streaming and Batch Generation

For larger cohorts, the engine supports both batched generation, in which patients are produced and written to disk in chunks, and parallel generation, in which multiple worker threads or processes simulate independent patients concurrently. Because each patient is statistically independent given the global seed, the simulation is embarrassingly parallel, and PySynthea uses `concurrent.futures.ThreadPoolExecutor` or `ProcessPoolExecutor` to scale across cores. The `threads` parameter in `GeneratorOptions` controls the level of parallelism, and the exporters are designed to write incrementally so that disk usage is bounded.

7.4 Integration with Dask, PySpark, and Airflow

Once generation has been wrapped as a Python function, integrating it with distributed orchestrators becomes straightforward. A Dask [15] or PySpark [16] driver can call the generator on each worker with a different seed and aggregate the resulting partitioned datasets. Workflow managers such as Apache Airflow [31] can schedule periodic regeneration of synthetic cohorts as part of a larger pipeline, for example to refresh test data nightly or to regenerate evaluation datasets when modules are updated. These integrations require no special support in PySynthea itself; they follow naturally from the fact that PySynthea is a library that lives in the same process model as the rest of the Python data ecosystem.

8 Healthcare Data Standards and Export Formats

Synthetic data is only as useful as the formats in which it can be consumed. PySynthea is designed to support the major healthcare data formats and to interoperate cleanly with general-purpose analytics formats.

Table 2: Summary of supported export formats and typical downstream uses.

Format	Output shape	Primary use
FHIR R4	One `Bundle` per patient, containing linked clinical resources	Interoperability testing, FHIR server loading, SMART-on-FHIR application development
CSV	One flat file per resource type, using Synthea-compatible identifiers	pandas analysis, spreadsheet review, database bulk loading
JSON	One nested document per patient	Document databases, debugging, and inspection of complete patient histories
Parquet	Columnar tables derived from pandas-compatible records	Data lakes, lakehouse storage, and distributed analytics with PySpark or Dask
Relational database	Tables written through SQLAlchemy or `pandas.to_sql`	Research warehouses, test databases, and downstream schema mappings

8.1 FHIR R4

The primary structured export format is FHIR R4 [24, 25]. Each simulated patient is rendered as a FHIR `Bundle` containing a `Patient` resource together with associated resources such as `Encounter`, `Condition`, `Observation`, `MedicationRequest`, `Procedure`, `Immunization`, `AllergyIntolerance`, `CarePlan`, `Device`, `ImagingStudy`, and `DiagnosticReport`. The exporter is configurable as either a transaction bundle or a collection bundle, and supports optional conformance with the US Core implementation guide. Resources reference each other using FHIR-standard relative URIs, so the resulting bundle can be ingested by any FHIR-conformant server, validated with standard FHIR tooling, and used in SMART-on-FHIR applications [28].

8.2 CSV

For flat-tabular consumers, PySynthea writes a directory of CSV files in the Synthea convention, with one file per resource type (e.g., `patients.csv`, `encounters.csv`, `conditions.csv`, `medications.csv`, `observations.csv`). These files are amenable to direct ingestion by pandas, by database bulk loaders, or by spreadsheet tools, and they preserve the joining keys necessary to reconstruct the longitudinal record.

8.3 JSON

A document-oriented JSON export is also provided, in which each patient is serialized as a single nested JSON document containing all of their attributes, encounters, and clinical events. This format is convenient for document databases, for NoSQL ingestion, and for hand-inspection during development.

8.4 Parquet and Data Lake Integration

Beyond the formats inherited from upstream Synthea, the architecture naturally supports columnar exports such as Apache Parquet [36] via pandas or Arrow. Because the in-memory representation is already pandas-compatible, a Parquet exporter is a thin wrapper around `DataFrame.to_parquet`. This makes PySynthea-generated data directly suitable for data lakes, lakehouse architectures, and downstream big-data analytics with PySpark.

8.5 Database Connectors

For users who prefer to write directly to a relational store, PySynthea-generated DataFrames can be written to a database through SQLAlchemy or `pandas.to_sql`. This allows synthetic cohorts to be loaded into research warehouses, OMOP-style schemas (via additional mapping), or test databases that mirror production schemas.

9 Performance and Scalability Considerations

Synthetic data generation is workload-heterogeneous: some users generate a handful of patients in an interactive notebook, while others generate millions for benchmarking or for pre-training experiments. PySynthea is designed to scale across this spectrum, with several mechanisms.

9.1 In-Process Performance

The single-patient critical path is dominated by state-machine evaluation across many timesteps. Within a single process, PySynthea minimizes per-step overhead by pre-parsing module definitions at load time (including converting raw code dictionaries to typed `Code` instances once), by capping module iterations to detect cycles in malformed modules, and by avoiding unnecessary deep copies of patient state. Where possible, attributes are kept in dictionary structures with O(1) access, and the simulation timestep is configurable through the properties file (the default is one week of simulated time).

9.2 Parallelism

Because patients are conditionally independent given their per-patient seeds, parallelism is straightforward. The generator exposes a `threads` parameter that controls a pool of worker threads or processes. For CPU-bound workloads, process-based parallelism via `ProcessPoolExecutor` avoids the GIL and scales to the number of physical cores. For I/O-bound workloads — typically dominated by export of large FHIR bundles to disk — thread-based parallelism is often sufficient and avoids the overhead of inter-process communication.

9.3 Memory Usage

For very large runs, PySynthea avoids holding the entire population in memory by writing each patient's exports incrementally as they finish. The generator's bookkeeping is dominated by per-batch summary statistics (totals of generated, living, and deceased patients, for example) and by the loaded module definitions, which are shared across patients.

9.4 Scaling Out

For scales beyond a single machine, the recommended pattern is to drive PySynthea from a distributed orchestrator such as Dask [15] or PySpark [16]. Each worker invokes the generator over a disjoint seed range, and the resulting partitioned outputs are aggregated into the target data store. Because the generator's outputs are file-based or DataFrame-based, this composition does not require any framework-specific support.

9.5 Cloud-Native Execution

The Python-native packaging also simplifies cloud deployment. A typical pattern is to package the generator as a container image, to run it inside a serverless or batch environment, and to write outputs directly to cloud object storage. Because the runtime is Python, this composition reuses the standard cloud Python toolchains, including SDKs for AWS, Google Cloud, and Azure, without bridging through JVM-specific patterns.

10 Example Workflows and Use Cases

This section illustrates how the design choices described above translate into concrete workflows.

10.1 Notebook-Based Cohort Generation

A typical workflow in a research notebook begins by importing the package, constructing a `GeneratorOptions` object, and invoking the generator. The user then loads the resulting FHIR bundles or CSVs into pandas and proceeds with analysis. Because the generator is in-process, intermediate state — including statistics about generated, living, and deceased patients — is available as ordinary attributes of the `Generator` object after the run finishes.

```
from synthea import Generator, GeneratorOptions

options = GeneratorOptions()
options.population_size = 200
options.state = "California"
options.seed = 42

generator = Generator(options)
generator.run()

print(generator.stats)  # e.g., {'total_generated': 200, 'living': 187, 'dead': 13, ...}
```

10.2 Custom Patient Construction

For pedagogical or debugging purposes, individual patients can be constructed explicitly and simulated outside of a full population run. The example below creates a single patient with fixed attributes and exports the resulting record to FHIR.

```
from datetime import datetime
from pathlib import Path
from synthea.world.person import Person
from synthea.engine.generator import Generator, GeneratorOptions
from synthea.export.fhir import FHIRExporter

person = Person(seed=12345)
person.attributes.update({
    "gender": "F",
    "birth_date": datetime(1980, 5, 15),
    "first_name": "Jane",
    "last_name": "Doe",
    "race": "white",
    "ethnicity": "non_hispanic",
})
person.init_health_record()

generator = Generator(GeneratorOptions())
generator._simulate_life(person)

exporter = FHIRExporter(generator.config, Path("./output"))
output_file = exporter.export(person, 0)
print(f"Exported to: {output_file}")
```

This pattern is useful for unit testing custom modules, for teaching, and for constructing minimal reproducible cases when debugging the simulation engine.

10.3 Parallel Batch Generation

For larger cohorts, the same API supports parallel generation. The example below generates patients across multiple US states in parallel using a thread pool, with deterministic seeds per state to ensure reproducibility.

```
import concurrent.futures
from synthea import Generator, GeneratorOptions

def generate_batch(state, size, seed):
    options = GeneratorOptions()
    options.population_size = size
    options.state = state
    options.seed = seed
    generator = Generator(options)
    generator.run()
    return generator.stats | {"state": state}

states = ["California", "Texas", "New York", "Florida"]
with concurrent.futures.ThreadPoolExecutor(max_workers=4) as ex:
    results = list(ex.map(
        lambda i_state: generate_batch(i_state[1], 100, i_state[0] * 1000),
        enumerate(states),
    ))
```

10.4 ML Benchmarking and Medical NLP

Because synthetic cohorts can be regenerated arbitrarily, they are well suited for benchmarking machine learning pipelines under controlled conditions. A researcher can fix a seed, generate a cohort, train a model, and rerun the same experiment elsewhere with bit-exact data. Cohorts can be parametrically varied — for example, by changing disease prevalence or the distribution of ages — to test model robustness. Medical NLP pipelines can be primed on synthetic discharge summaries or structured records before being applied to real, smaller, protected corpora [1, 19].

10.5 Federated Learning Simulation

For federated learning experiments [37], synthetic cohorts can stand in for the per-institution shards of a federation. Each shard can be parameterized differently to reflect institutional heterogeneity, and the resulting setup can be used to study algorithmic properties of federated training without requiring access to data from any real institution.

10.6 Education and Healthcare Interoperability Testing

In educational settings, PySynthea allows instructors to distribute a single notebook that produces a known, reproducible cohort, and to build entire lecture sequences around it. In software development settings, the same generator can produce a controlled set of FHIR bundles for end-to-end interoperability tests, replacing brittle hand-crafted fixtures and ensuring that test data covers a wide range of realistic clinical scenarios.

11 Discussion

The design of PySynthea trades certain properties of the original Java-based Synthea against others that are more important in modern Python-based workflows. We discuss these tradeoffs explicitly.

11.1 Fidelity Versus Accessibility

The first tradeoff is between absolute fidelity to the upstream Synthea reference implementation and the ergonomic accessibility of a clean Python rewrite. Because PySynthea consumes the same GMF JSON modules unchanged, it preserves the bulk of the clinical content that gives Synthea its value. However, the engine itself is an independent reimplementation, and small semantic differences — for example, in how rare edge cases in transitions or guards are evaluated — are possible. We treat the upstream Java implementation as the semantic reference and aim to converge with it on shared modules, with explicit divergence tracked in the project's documentation.

11.2 Statistical Realism

Like upstream Synthea, PySynthea generates patients whose statistical properties are calibrated against external references (census demographics, growth charts, cost data, immunization schedules) but whose joint statistical structure does not faithfully reproduce that of any particular real cohort. Synthetic Synthea-style data is therefore useful for structural and pipeline-level applications — schema validation, interoperability testing, model engineering, education — and for many forms of methodological experimentation, but it should not be confused with a generative model trained on real data, such as a generative adversarial network or a diffusion model applied to EHRs [29, 38]. We see these two classes of synthetic data as complementary rather than competing: Synthea-style data provides clinically structured longitudinal records with full provenance, while learned generative models provide higher fidelity to specific distributions at the cost of interpretability and standards compliance.

11.3 Python Performance Considerations

A common concern about Python-native reimplementations of CPU-intensive workloads is performance. In our experience, the per-patient simulation cost is dominated by state-machine evaluation and by serialization, both of which can be optimized within Python through pre-parsing, simple data structures, and efficient JSON or Parquet writers. For workloads beyond what a single Python process can sustain, the recommended path is horizontal scaling via processes, Dask, or PySpark, rather than vertical optimization of a single process. Future versions may use Cython, Numba, or Rust-based extensions to accelerate the inner loop of state-machine evaluation if profiling indicates that this is the binding constraint.

11.4 Long-Term Maintenance

Reimplementing a substantial open-source project always raises questions about long-term maintenance. PySynthea is explicitly designed to consume upstream Synthea's module library, so improvements to clinical content there flow into PySynthea without code-level synchronization. The engine itself is more compact than the full Java codebase, and is organized so that contributors

familiar with Python and basic state-machine concepts can extend it without needing to learn a new toolchain.

11.5 Compatibility with Upstream Synthea

A core design constraint is that PySynthea should remain a "good citizen" within the broader Synthea ecosystem. This shows up in three places. First, PySynthea consumes the same GMF JSON modules, so the shared community library of disease content remains the single source of truth. Second, PySynthea produces FHIR bundles, CSVs, and JSON documents that are compatible at the schema level with upstream Synthea, so existing consumers — FHIR servers, validation pipelines, analytics tooling — can ingest PySynthea outputs without modification. Third, PySynthea adopts the same configuration model and many of the same CLI options, so users moving between the two implementations encounter a familiar interface.

12 Future Work

PySynthea opens several lines of future research and engineering work, each of which builds on the Python-native foundation described above.

GPU acceleration. The inner loop of state-machine evaluation, particularly when run over millions of patients, is a candidate for GPU acceleration. While the current implementation uses CPU-based parallelism, longer-term work may explore vectorized or batched simulation strategies that exploit GPUs through CUDA, JAX, or PyTorch, especially for the subset of states whose semantics are amenable to batched evaluation.

LLM-generated and LLM-assisted disease modules. Large language models have demonstrated significant capability in authoring structured clinical content [18, 19]. A natural extension is to use LLMs to draft GMF modules from clinical guidelines or literature, with human-in-the-loop review. PySynthea's Python-native nature makes it straightforward to embed such workflows in the same environment used to train or evaluate the LLMs themselves.

Multi-agent patient simulation. Synthea's current model treats patients as independent, but realistic healthcare populations include interaction effects — outbreaks, household transmission, healthcare system congestion — that are best modeled with multi-agent simulation. Extending PySynthea with multi-agent dynamics, where individual patient simulations are coupled through shared environmental state, opens new research possibilities in infectious disease modeling and healthcare operations research.

Reinforcement learning environments. The discrete-time, state-driven nature of Synthea makes it a natural substrate for reinforcement learning environments grounded in clinical decision making. A PySynthea-backed environment that exposes patient state to an agent and accepts treatment actions could serve as a benchmark for offline and online RL in clinical care, while keeping the underlying data fully synthetic.

Differential privacy and privacy-preserving generation. Although Synthea-style data does not derive from real patients and therefore does not require privacy guarantees in the traditional sense, the framework can serve as a useful testbed for differentially private machine learning algorithms [39] and for evaluating membership inference risk in models trained on simulated cohorts.

Synthetic imaging integration. Modern healthcare data is increasingly multimodal. Integrating PySynthea with synthetic imaging pipelines — for example, generative models for chest X-rays

or pathology images linked to simulated clinical histories — would enable end-to-end multimodal benchmarks that combine structured EHR data with synthetic medical images.

13 Conclusion

Synthetic healthcare data is an increasingly important resource for research, education, software engineering, and applied machine learning in medicine. By generating realistic longitudinal records without disclosing information about real individuals, it sidesteps the regulatory and operational constraints that limit access to real EHRs and provides a substrate for reproducible, shareable, and scalable experimentation. Synthea has established itself as a leading tool in this space, with a state-machine-based simulation model, a rich library of disease modules, and broad support for healthcare interoperability standards [20, 30].

The friction in current workflows, however, is no longer about the quality of the synthetic data itself. It is about the alignment between the data generator and the dominant data science ecosystem. Python is now the de facto standard for data analysis, machine learning, and AI in healthcare and elsewhere, and tools that require external runtimes or file-based bridges into Python pay a real cost in adoption, in reproducibility, and in the velocity of research.

PySynthea addresses this gap by reimplementing the Synthea engine as a pure-Python, pip-installable framework that preserves the Generic Module Framework, supports standards-conformant export formats, and integrates naturally with pandas, NumPy, PyTorch, Dask, PySpark, Airflow, and Jupyter. It is designed to be accessible from the first line of a notebook, reproducible across runs, extensible by researchers in their native language, and scalable from interactive sessions to distributed batch jobs. By lowering the cost of adopting synthetic healthcare data and by aligning the generator with the rest of the modern data science toolchain, PySynthea aims to broaden participation in synthetic-data-driven research and to accelerate the responsible development of machine learning systems for medicine.

Acknowledgements

The authors thank the TietAI clinical team for feedback on governance requirements and the Hydra Platform engineering team for integration support.

References

[1] Alvin Rajkomar, Eyal Oren, Kai Chen, Andrew M. Dai, Nissan Hajaj, Michaela Hardt, Peter J. Liu, Xiaobing Liu, Jake Marcus, Mimi Sun, et al. Scalable and accurate deep learning with electronic health records. *npj Digital Medicine*, 1(1):18, 2018.

[2] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature Medicine*, 25(1):24–29, 2019.

[3] Eric J. Topol. High-performance medicine: the convergence of human and artificial intelligence. *Nature Medicine*, 25(1):44–56, 2019.

[4] Khaled El Emam, Sam Rodgers, and Bradley Malin. Anonymising and sharing individual patient data. *BMJ*, 350:h1139, 2015.

[5] Kathleen Benitez and Bradley Malin. Beyond safe harbor: Automatic discovery of health information de-identification policy alternatives. *Journal of the American Medical Informatics Association*, 17(6):706–712, 2010.

[6] Richard J. Chen, Ming Y. Lu, Tiffany Y. Chen, Drew F.K. Williamson, and Faisal Mahmood. Synthetic data in machine learning for medicine and healthcare. *Nature Biomedical Engineering*, 5(6):493–497, 2021.

[7] Aldren Gonzales, Guruprabha Guruswamy, and Scott R. Smith. Synthetic data in health care: A narrative review. *PLOS Digital Health*, 2(1):e0000082, 2023.

[8] U.S. Department of Health and Human Services. Health insurance portability and accountability act of 1996. Public Law 104–191, 1996.

[9] European Parliament and Council. Regulation (EU) 2016/679 of the european parliament and of the council on the protection of natural persons with regard to the processing of personal data (GDPR). Official Journal of the European Union, L119, 2016.

[10] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[11] Wes McKinney. Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56, 2010.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, pages 130–136, 2015.

[16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, et al. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[18] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020.

[19] Michael Wornow, Yizhe Xu, Rahul Thapa, Birju Patel, Ethan Steinberg, Scott Fleming, Michael A. Pfeffer, Jason Fries, and Nigam H. Shah. The shaky foundations of large language models and foundation models for electronic health records. *npj Digital Medicine*, 6:135, 2023.

[20] Jason Walonoski, Mark Kramer, Joseph Nichols, Andre Quina, Chris Moesel, Dylan Hall, Carlton Duffett, Kudakwashe Dube, Thomas Gallagher, and Scott McLachlan. Synthea: An approach, method, and software mechanism for generating synthetic patients and the synthetic electronic health care record. *Journal of the American Medical Informatics Association*, 25(3):230–238, 2018.

[21] Astral Software. uv: An extremely fast Python package and project manager. https://github.com/astral-sh/uv, 2024.

[22] The MITRE Corporation. Synthea: Synthetic patient population simulator. https://github.com/synthetichealth/synthea, 2024. Accessed: 2026-05-10.

[23] Kudakwashe Dube, Scott McLachlan, Jason Walonoski, Sarah Tashman, and Mark Kramer. Implementing a generic framework for the simulation of synthetic patient populations and electronic health records. *Studies in Health Technology and Informatics*, 281:123–127, 2021.

[24] Duane Bender and Kamran Sartipi. HL7 FHIR: An agile and RESTful approach to healthcare information exchange. In *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems*, pages 326–331. IEEE, 2013.

[25] HL7 International. HL7 FHIR Release 4. https://www.hl7.org/fhir/R4/, 2019. Accessed: 2026-05-10.

[26] SNOMED International. SNOMED CT: Systematized nomenclature of medicine – clinical terms. https://www.snomed.org/, 2024.

[27] Regenstrief Institute. LOINC: Logical observation identifiers names and codes. https://loinc.org/, 2024.

[28] Joshua C. Mandel, David A. Kreda, Kenneth D. Mandl, Isaac S. Kohane, and Rachel B. Ramoni. SMART on FHIR: A standards-based, interoperable apps platform for electronic health records. *Journal of the American Medical Informatics Association*, 23(5):899–908, 2016.

[29] Chao Yan, Yao Yan, Zhiyu Wan, Ziqi Zhang, Larsson Omberg, Justin Guinney, Sean D. Mooney, and Bradley A. Malin. A multifaceted benchmarking of synthetic electronic health record generation models. *Nature Communications*, 13:7609, 2022.

[30] Jason Walonoski, Stephen Klaus, Eldesia Granger, Dylan Hall, Andrew Gregorowicz, George Neyarapally, Abigail Watson, and Jeff Eastman. Synthea novel coronavirus (covid-19) model and synthetic data set. *Intelligence-Based Medicine*, 1:100007, 2020.

[31] Apache Software Foundation. Apache Airflow. https://airflow.apache.org/, 2024.

[32] Alistair E.W. Johnson, Tom J. Pollard, Lu Shen, Li-wei H. Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G. Mark. MIMIC-III, a freely accessible critical care database. *Scientific Data*, 3:160035, 2016.

[33] Alistair E.W. Johnson, Lucas Bulgarelli, Lu Shen, Alvin Gayles, Ayad Shammout, Steven Horng, Tom J. Pollard, Sicheng Hao, Benjamin Moody, Brian Gow, et al. MIMIC-IV, a freely accessible electronic health record dataset. *Scientific Data*, 10:1, 2023.

[34] Tom J. Pollard, Alistair E.W. Johnson, Jesse D. Raffa, Leo A. Celi, Roger G. Mark, and Omar Badawi. The eICU collaborative research database, a freely available multi-center database for critical care research. *Scientific Data*, 5:180178, 2018.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[36] Apache Software Foundation. Apache Parquet. https://parquet.apache.org/, 2024.

[37] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics (AISTATS)*, pages 1273–1282, 2017.

[38] Edward Choi, Siddharth Biswal, Bradley Malin, Jon Duke, Walter F. Stewart, and Jimeng Sun. Generating multi-label discrete patient records using generative adversarial networks. In *Machine Learning for Healthcare Conference (MLHC)*, pages 286–305, 2017.

[39] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 1–12. Springer, 2006.