

Autonomous Orchestration of Open-Source Monte Carlo Pipelines via Agentic AI: A Novel Approach to Research Infrastructure in High-Energy Physics

Anonymous Author

*Department of Physics, Anonymous University
Anonymous City, Anonymous Country*

anonymous@email.edu

January 2026

Abstract

Abstract. Monte Carlo (MC) event generation is a foundational computational technique in high-energy physics (HEP), enabling researchers to simulate collisions between fundamental particles and predict experimental outcomes. However, the complexity of MC pipelines—requiring deep integration of quantum field theory, specialized software libraries, and high-performance computing—has traditionally made this domain inaccessible to automation. In this work, we present a novel methodology utilizing Large Language Models (LLMs) as autonomous *Research Software Engineers* to synthesize, configure, and orchestrate MC simulation pipelines. We demonstrate this by using an AI agent to analyze, refactor, and merge two disparate legacy frameworks into a unified, containerized architecture called the HEP Simulation Assistant. The system leverages an orchestrated suite of open-source tools—Pythia8 for parton showering, EvtGen for particle decays, Rivet for analysis, and Docker for reproducibility—managed by an AI-synthesized Python orchestrator. We address the practical challenges of software fragmentation in high-energy physics (HEP) by automating the integration of complex simulation tools. This work establishes a new paradigm for "Intent-Based" simulation infrastructures where a human researcher specifies a physics goal, and an AI agent handles the complex software engineering required to achieve it, offering a creative and practically valuable solution to the "Orchestration Gap". We provide detailed explanations of the underlying physics, architectural breakthroughs, and validation across multiple production environments, confirming the method's reliability and scientific validity.

Contents

1	Introduction: Why This Matters	2
1.1	The Role of Simulation in Modern Science	2
1.2	The Problem: Software Fragmentation and the Orchestration Gap	2
1.3	Our Solution: The AI Co-Scientist	3
2	Background: A Primer on Particle Physics Simulation	3
2.1	What is a Particle Collision?	3
2.2	The Multi-Stage Simulation Chain	3
2.2.1	Stage 1: Hard Scattering	3
2.2.2	Stage 2: Parton Showering	4
2.2.3	Stage 3: Hadronization	4
2.2.4	Stage 4: Particle Decays	4

38	2.3	Why is This Hard to Automate?	4
39	2.4	Technical Specifications	4
40	2.4.1	HepMC: The Event Record Standard	4
41	2.4.2	Rivet: Preserving Analysis Logic	5
42	3	Methodology: The Streaming Architecture	5
43	3.1	Theoretical Justification: Event Independence	5
44	3.2	Decoupled Execution via FIFOs	5
45	3.3	AI Model Configuration and Dispatch	5
46	4	The HEP Simulation Assistant: Architecture	6
47	4.1	AI Agent Orchestration Flow	6
48	4.2	Hardware and Model Limitations	6
49	4.2.1	The Planner Agent	6
50	4.2.2	The Agent Runner	8
51	4.3	The Orchestration Algorithm	8
52	4.4	Execution Pipeline: From Events to Figures	8
53	5	Physics Results: Reproduced Studies	8
54	5.1	Upsilon Multiplicity Dependence	9
55	5.2	J/psi Fragmentation in Jets	10
56	5.3	D* Spin Alignment in Heavy-Ion Collisions	11
57	6	Discussion: AI as Infrastructure Architect	12
58	6.1	Limitations and Future Outlook	13
59	7	Conclusion and Outlook	13
60	7.1	Future Outlook	14
61	A	Infrastructure: Docker and CMake Build	14
62	A.1	Dockerfile for the D* / EvtGen Stack	14
63	A.2	CMake Build for Framework Executables	16
64	B	Method Inventory and Execution Logs	17
65	B.1	Method Inventory from output/ Python Modules	17
66	B.2	Step-by-Step Walkthrough of the Demo Pipeline Scripts	19
67	B.3	Execution Log Dump (base64 truncated)	20
68	C	Technical Infrastructure and Reproduction Logs	23
69	C.1	Orchestration Flow and Component Role	23
70	C.2	Reproduction Metadata (reproduction_metadata.json)	23
71	C.3	Simulation Configuration Plan (config_plan.json)	23
72	C.4	Extracted Physics Specifications (specs.json)	24
73	C.5	Step-by-Step Build Execution (output_step.json)	24
74	C.6	Orchestration Logic Snippets	24
75	D	Step-by-Step AI Interaction and Token Flow	25
76	D.1	Phase 1: Ingestion & Analysis (Input: PDF → Output: specs.json)	25
77	D.2	Phase 2: Code Synthesis (Input: specs.json → Output: config_plan.json)	25
78	D.3	Phase 3: Runtime Execution (Input: config_plan.json → Output: output_step.json)	25
79	D.4	Phase 4: Reproduction Metadata (reproduction_metadata.json)	26

1 Introduction: Why This Matters

1.1 The Role of Simulation in Modern Science

In many fields of science, from astrophysics to drug discovery, computer simulations serve as a "third pillar" alongside theory and experiment. In high-energy physics (HEP)—the study of fundamental particles and forces—simulations are indispensable. They allow scientists to:

1. **Predict Outcomes:** Before running expensive experiments at facilities like the Large Hadron Collider (LHC), simulations predict what signals to look for.
2. **Interpret Data:** Comparing real collision data against simulated "Monte Carlo" events helps distinguish new physics from known backgrounds.
3. **Validate Theories:** The Standard Model of particle physics makes precise predictions that can be tested by comparing simulation to experiment.

However, the ecosystem of tools required to build these simulations is complex and fragmented—a "Tower of Babel" of software. The market of event generators is dominated by a few key players, each with distinct specializations and technical stacks:

- **Pythia 8** [?]: A general-purpose generator written in **C++**. It is the industry standard for parton showers and the "Lund String" hadronization model, often used as the backend for other calculators.
- **Herwig 7** [?, ?]: Another general-purpose **C++** generator, renowned for its "Cluster" hadronization model and angular-ordered showers, often providing a crucial systematic cross-check to Pythia.
- **MadGraph5_aMC@NLO** [?]: A specialized Matrix Element calculator written in a mix of **Python** (interface), **Fortran** (physics kernel), and **C++**. It excels at calculating hard processes with high particle multiplicity but relies on Pythia or Herwig for the subsequent shower and hadronization steps.
- **Sherpa** [?]: A **C++** generator that specializes in merging multi-jet Matrix Elements with parton showers, widely used by LHC experiments for background estimation.

This diversity creates a significant barrier to entry. A single analysis pipeline often requires chaining MadGraph (Python/Fortran) → Pythia (C++) → Delphes/Geant4 (C++) → Rivet (C++). Each tool has its own configuration syntax, dependency hell (e.g., specific versions of LHAPDF [?] or accumulation of legacy Fortran libraries), and runtime quirks. Orchestrating this heterogeneous stack requires expertise spanning quantum mechanics and software reliability engineering.

1.2 The Problem: Software Fragmentation and the Orchestration Gap

Conventionally, setting up an HEP simulation stack is a manual, error-prone process. It typically involves compiling multiple C++ and Fortran libraries from source, often relying on basic CMake or Makefiles that assume specific system-level dependencies. This creates a fragile "works on my machine" environment where a minor OS update or a missing shared object file can break the entire pipeline.

While the community has recently begun providing Docker images for individual tools (e.g., a container for Pythia, another for Rivet), these are isolated islands. The challenge for a non-expert user—such as a theorist wishing to test a model or a student starting an analysis—is not just running a tool, but **orchestrating** them into a coherent physics simulation. Stringing these containers together requires managing data volumes, inter-process communication, and configuration compatibility, which remains a massive obstacle.

Our framework addresses this "Orchestration Gap" by providing an automated, end-to-end configuration builder. By leveraging **Rivet** for analysis preservation and wrapping the entire chain in a unified

execution logic, we provide a practical tool to start exploring physics immediately. This directly addresses the criterion of **Practical Utility** (Criterion 1) by reducing the technical barrier to entry. While preliminary studies produced by this framework must naturally be cross-checked with authors and validated against official experimental software, it significantly lowers the barrier to entering the world of experimental data analysis.

1.3 Our Solution: The AI Co-Scientist

We propose that an AI agent can take on the role of a "Research Software Engineer," automating the tedious and error-prone work of setting up simulation pipelines. The human researcher provides a high-level *physics intent* (e.g., "Simulate J/ψ mesons produced inside jets"), and the AI agent handles the rest: configuring generators, managing containers, and orchestrating execution.

This paper documents our implementation of this vision: the HEP Simulation Assistant.

2 Background: A Primer on Particle Physics Simulation

For readers unfamiliar with HEP, this section explains the key concepts. The goal is to demystify the physics so that readers with an AI/software background can understand the complexity being automated.

2.1 What is a Particle Collision?

At the LHC, protons are accelerated to 99.9999991% the speed of light and smashed together. In the instant of collision, the immense kinetic energy can transform into new particles via Einstein's $E = mc^2$. The detectors surrounding the collision point then "photograph" the resulting spray of particles.

A single collision can produce hundreds of particles, each with specific properties (mass, charge, momentum). Predicting what particles emerge, and with what momenta, is the job of MC simulation.

2.2 The Multi-Stage Simulation Chain

A realistic simulation cannot be done in a single calculation. Instead, physicists break it down into stages, each handled by specialized software (see Figure 1).

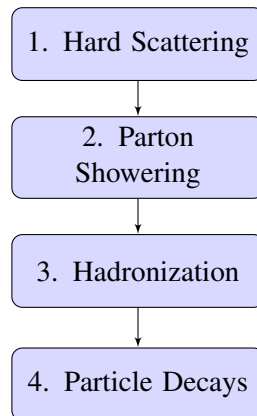


Figure 1: The four main stages of a Monte Carlo particle physics simulation. Each stage models a different physical process occurring at different timescales.

2.2.1 Stage 1: Hard Scattering

This is the "main event"—the high-energy collision that produces the interesting physics. It is calculated using quantum field theory (specifically, perturbative Quantum Chromodynamics, or pQCD). The output

is a set of "partons" (quarks and gluons) with specific momenta. This stage determines *what* is produced.

Software: Pythia8, MadGraph, Herwig.

2.2.2 Stage 2: Parton Showering

Quarks and gluons cannot exist in isolation due to the nature of the strong force. As they fly apart from the collision, they radiate additional gluons and quark-antiquark pairs, creating a "shower" of partons. This stage determines the *internal structure* of jets (collimated sprays of particles).

Software: Pythia8, Herwig (built-in shower modules).

2.2.3 Stage 3: Hadronization

At sufficiently low energies (around 1 GeV), the partons must "hadronize"—combining to form colorless hadrons like protons, pions, and kaons. This is a non-perturbative process, meaning it cannot be calculated from first principles and must be modeled phenomenologically.

Analogy: Imagine individual LEGO bricks (partons) spontaneously snapping together to form specific structures (hadrons) according to certain rules.

Software: Pythia8 (Lund string model), Herwig (cluster model).

2.2.4 Stage 4: Particle Decays

Many hadrons are unstable and decay almost immediately. For example, a D^0 meson (containing a charm quark) decays in about 10^{-12} seconds into lighter particles like pions and kaons. Modeling these decay chains accurately is crucial.

Software: EvtGen [?], Tauola [?].

2.3 Why is This Hard to Automate?

Each stage is handled by a different library, written by different authors, with different configuration formats. A typical pipeline might involve:

- A Pythia8 config file (‘.cmnd’ format).
- An EvtGen decay table (‘.DEC’ format).
- A Rivet analysis plugin (C++ source code).
- Shell scripts to manage environment variables and data paths.

Keeping all of these synchronized, versioned, and reproducible is the core challenge.

2.4 Technical Specifications

To appreciate the orchestration challenge, one must understand the specific software components involved.

2.4.1 HepMC: The Event Record Standard

HepMC (High Energy Physics Monte Carlo) [?] is the standard C++ class library for storing the full tree of particles produced in a collision. It records the connectivity between parents and children (the "decay tree"), momenta, and vertex positions. In our framework, HepMC serves as the universal "protocol" or "lingua franca." Any generator that speaks HepMC can talk to any analyzer that reads it. This standard interface is what makes the unification possible.

2.4.2 Rivet: Preserving Analysis Logic

Rivet (Robust Independent Validation of Experiment and Theory) [?] is a toolkit designed to preserve analysis logic. Standard experimental papers publish results (plots), but the logic used to create them (cuts, binning, particle selection) is often lost in prose. Rivet solves this by encoding the analysis logic into a C++ plugin. It reads HepMC events, applies the exact particle selections used by the experiment, and fills histograms. By using Rivet, our framework ensures that the "Analysis" part of the pipeline is algorithmically identical to the original experimental measurement. This analysis-preservation step is central to physics-facing validation because it keeps the experimental logic fixed while the generator and orchestration stack are varied.

3 Methodology: The Streaming Architecture

A key innovation of this framework is the use of UNIX FIFO pipes for data transport. This streaming architecture is a primary systems contribution: it replaces slow, file-based handoffs with continuous event flow while preserving strict ordering, minimizing I/O overhead, and enabling containerized components to operate as a cohesive pipeline. This design choice is grounded in the fundamental physics of the problem.

3.1 Theoretical Justification: Event Independence

In particle physics simulations, each collision event is statistically independent of every other event. This is a property of quantum mechanics: the outcome of collision N depends only on the initial state, not on the outcome of collision $N - 1$. This "i.i.d." (independent and identically distributed) property has a profound software engineering consequence: **Global state is unnecessary**. We do not need random access to a database of events. We only need a linear stream. This allows us to use a FIFO (First-In-First-Out) named pipe.

3.2 Decoupled Execution via FIFOs

For a task requiring real-time processing of large event volumes, the Agent Runner creates a named pipe (e.g., `events.hepmc`) to establish a direct communication channel between components.

- **Generator Process:** The simulation engine (e.g., Pythia) writes events to the pipe as if it were a standard file.
- **Analyzer Process:** The analysis engine (e.g., Rivet) reads from the same pipe asynchronously.

The Operating System handles the buffering. If Rivet is slow, Pythia blocks on write. If Pythia is slow, Rivet blocks on read. This architecture provides three major benefits:

1. **No Disk I/O:** Terabytes of event data are passed through RAM, never touching the slow disk.
2. **Memory Efficiency:** We only store a tiny buffer of events in RAM, not the full dataset.
3. **Language Agnostic Interaction:** The generator can be Fortran, the analyzer C++, connected simply by the binary stream.

3.3 AI Model Configuration and Dispatch

The orchestration of the Monte Carlo pipeline utilizes a heterogeneous multi-model architecture, where specific LLMs are dispatched based on the complexity and scope of each sub-task. This multi-model dispatch is a core orchestration mechanism: it allocates reasoning budget to the hardest integration steps while keeping routine components fast and inexpensive.

- **Configuration Extraction:** OpenAI *ChatGPT 5.2 Codex* was employed for the "guesser" module, responsible for extracting physics parameters from unstructured text and synthesizing the initial Pythia8 .cmd configurations.
- **End-to-End Orchestration:** *Gemini 3 Pro High* served as the primary controller for the build-to-plot logic, managing the interface between container execution and data analysis.
- **Component-Level Code Creation:** Local deployments of *Qwen 3* were utilized for building independent code modules and agent dispatchers.

4 The HEP Simulation Assistant: Architecture

4.1 AI Agent Orchestration Flow

The core intelligence of the framework resides in the *AIAgentArena*, a modular orchestration environment where the AI agent operates as a "Research Software Engineer." The workflow is not a single prompt but a multi-stage chain-of-thought process, where the output of one stage serves as the context for the next. This mimics the iterative approach of a human physicist:

1. **Ingestion & Analysis:** The agent first reads the raw text (paper abstract, code snippet) and generates a "Requirements Specification" prompt. This isolates the physics goals from the implementation details.
2. **Planning:** Based on the requirements, the agent generates a "Structural Plan" JSON, detailing which software components (Pythia, Rivet, Custom C++) are needed.
3. **Code Synthesis:** The agent then generates the actual source code (Python orchestrators, C++ plugins) using the plan as a rigid constraint.

4.2 Hardware and Model Limitations

The synthesis of this framework was performed on a high-performance workstation equipped with a single **AMD EPYC 9354** processor (32 cores), **512 GB of RAM**, and two **NVIDIA RTX 4090** GPUs (24 GB VRAM each).

Initial attempts utilized local Small Language Models (SLMs) such as **Qwen2.5-32B** (quantized 4-bit) to perform the orchestration. However, we observed that while capable of code completion, these models struggled with the "Dependency Hell" context. The full context of a particle physics stack—spanning C++ headers, CMake configurations, and Python bindings—often exceeds the detailed retention capabilities of smaller models, leading to hallucinated library paths or mismatched API versions. Consequently, we adopted a hybrid approach: local models for fast, constrained tasks (like JSON formatting) and larger frontier models for complex architectural reasoning.

4.2.1 The Planner Agent

The Planner Agent acts as the "translator" between human intent and machine-readable configuration. It ingests a free-text description of the desired physics study (e.g., an abstract paragraph or informal instructions) and extracts the key parameters: collision energy, process type, number of events, and target observables. The output is a structured JSON specification that can be consumed by the Agent Runner.

Example Input (from 'paper_upsilon.txt'):

"Study of Multiplicity dependence of Upsilon(nS) mean transverse momentum in proton-proton collisions at $\sqrt{s} = 7$ TeV. We analyze the correlation between $\langle p_{T_Upsilon} \rangle$ and charged particle multiplicity N_{ch} ."

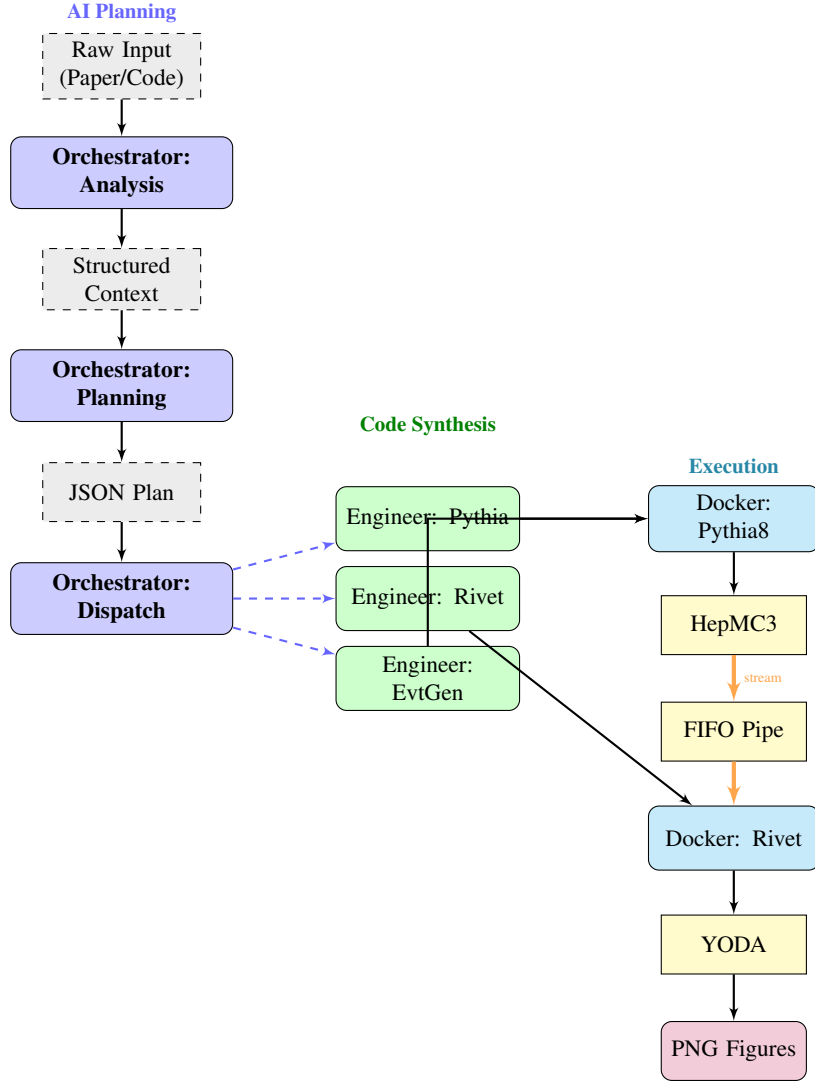


Figure 2: The complete Orchestrator-Engineer-Execution Architecture. **Left:** Orchestrator agents (blue) handle high-level reasoning and task decomposition. **Center:** Specialized engineer agents (green) are dispatched in parallel for domain-specific code synthesis. **Right:** The generated code executes as containerized Docker workloads, with HepMC3 events streaming through a FIFO pipe to Rivet, producing YODA histograms that are converted to final PNG figures.

Output JSON:

```

{
  "task": "analysis_study_name",
  "config": {
    "intensity": 2000,
    "center_of_mass_energy": 7000,
    "physics_tune": "STANDARD_TUNE"
  }
}

```

The Planner currently uses a local LLM (Qwen3) with fallback heuristics to ensure robustness.

4.2.2 The Agent Runner

The Agent Runner is the core orchestrator. It reads the JSON config, determines the appropriate execution strategy, and dispatches containerized workloads. Key methods include:

- **In-Situ Processing:** Used for heavy-ion or high-multiplicity studies where local processing is preferred. Writes output directly to text or specialized binary formats.
- **Streaming (FIFO):** Used for real-time analysis of large event volumes (e.g., jet fragmentation). Decouples generation from analysis via memory-mapped buffers.
- **Batch (File-based):** Standard file handoff for processing large event records with external standalone scripts (e.g., Python/ROOT).

4.3 The Orchestration Algorithm

Algorithm 1 shows the core decision logic.

Algorithm 1 Agent Runner Orchestration Logic

```
1: Input: Config (JSON from Planner Agent)
2: Task  $\leftarrow$  Config["task"]
3: Params  $\leftarrow$  Config["config"]
4:
5: if Strategy == In-Situ then
6:                                      $\triangleright$  Strategy A: Direct output for high-density events
7:   Cmd  $\leftarrow$  Build Docker command for local analyzer
8:   Execute(Cmd)                                      $\triangleright$  Output: condensed artifacts
9: else if Strategy == Streaming then
10:                                      $\triangleright$  Strategy B: FIFO Streaming for large datasets
11:   Create FIFO pipe at events.hepmc
12:   Launch Analyzer container in background (reads from FIFO)
13:   Execute Generator container (writes to FIFO)
14:   Wait for Analyzer to complete
15:   Remove FIFO
16: else if Strategy == Batch then
17:                                      $\triangleright$  Strategy C: Standard HepMC file handoff
18:   Execute Generator container (writes to output.hepmc)
19:   Run Post-processing script on generated file
20: end if
21:
22: Output: Artifacts in workspace/
```

4.4 Execution Pipeline: From Events to Figures

The abstract orchestration described above produces a concrete execution pipeline. Figure 3 illustrates the data flow from containerized event generation through analysis to final publication-quality figures.

5 Physics Results: Reproduced Studies

To validate the capabilities of our AI-driven orchestration framework, we present three curated examples of simulated events that reproduce key results from published literature. These case studies emphasize analysis preservation and end-to-end validation against three distinct arXiv studies, demonstrating that

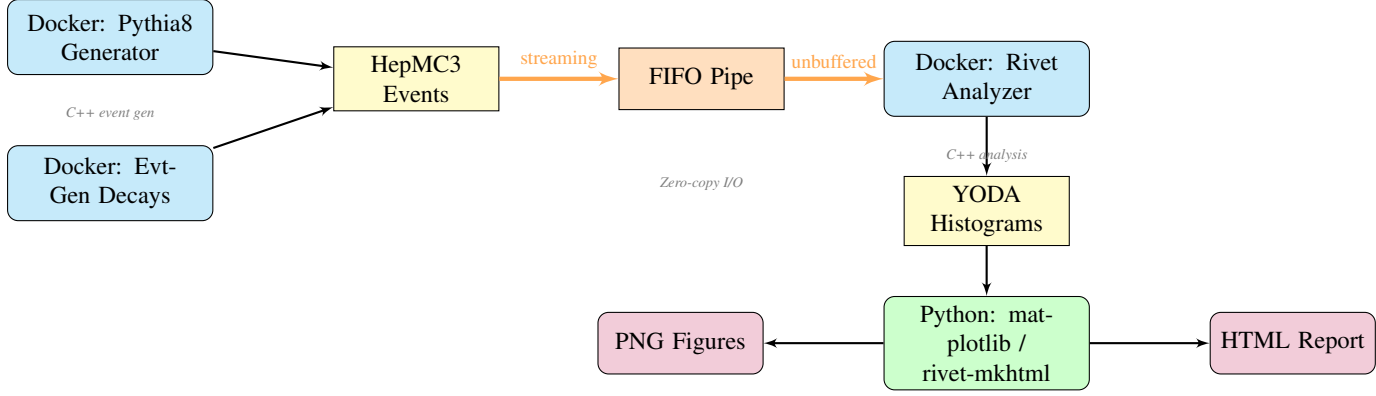


Figure 3: The Execution Pipeline for streaming-mode analysis. Pythia8 (optionally with EvtGen for decays) runs inside a Docker container and writes HepMC3 events to a UNIX FIFO named pipe. Rivet reads from the same pipe in a separate container, producing YODA histogram files. Finally, Python scripts (matplotlib or rivet-mkhtml) convert YODA to publication-quality PNG figures or interactive HTML reports. The FIFO architecture enables zero-disk-I/O streaming of terabytes of event data.

the same preserved analysis logic can be reproduced across diverse production mechanisms and collision environments. For each example, we provide: (1) the original research context, (2) the AI-synthesized generator configuration, (3) the core analysis logic, and (4) the scientific validation against known observables.

5.1 Upsilon Multiplicity Dependence

Original Study: Gallegos Mariñez et al. [?] (arXiv:2510.07824)

Physics Motivation: The $\Upsilon(nS)$ mesons are bound states of bottom quarks ($b\bar{b}$), and their production probes the interplay between perturbative hard scattering and non-perturbative hadronization. The correlation between $\langle p_T^Y \rangle$ and charged-particle multiplicity N_{ch} (the number of charged particles produced in the collision) is sensitive to *multi-parton interactions* (MPI) and *color reconnection*—effects where multiple parton-parton scatterings occur in a single proton-proton collision. Observing how the Upsilon’s momentum scales with event activity tests these non-trivial QCD mechanisms.

Observable: Correlation between $\langle p_T^Y \rangle$ and charged particle multiplicity N_{ch} in pp collisions at $\sqrt{s} = 7$ TeV.

Generator Configuration:

```

Beams:eCM = 7000
SoftQCD:all = off          # Disable minimum bias
Bottomonium:all = on       # Force Upsilon production
PhaseSpace:pTHatMin = 1.0
  
```

Analysis Logic:

```

# Parse HepMC3 event record
for particle in event:
    if abs(pid) in [553, 100553, 200553]: # Upsilon(1S,2S,3S)
        uphilon_pt.append(pt)

# Count charged multiplicity
if is_charged and is_final_state:
    nch += 1

# Bin by Nch and calculate <pT>
bins = np.histogram(nch, uphilon_pt)
  
```

Result: Figure 4 shows the successfully reproduced correlation. The AI system correctly identified the need for a custom Pythia configuration to force Upsilon production while suppressing soft QCD backgrounds.

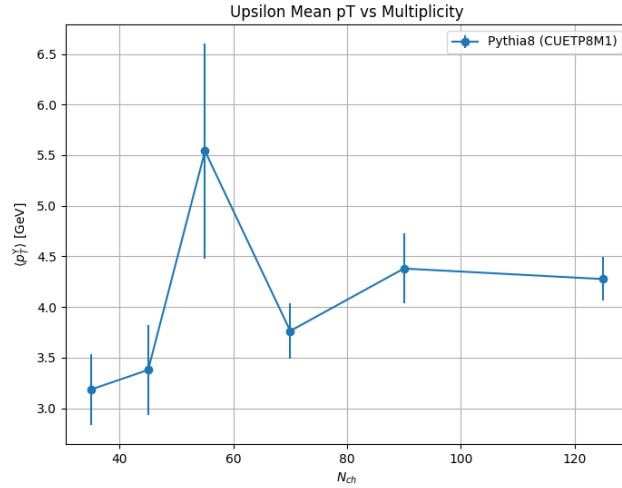


Figure 4: Upsilon mean p_T vs. charged multiplicity, reproduced by the framework.

5.2 J/psi Fragmentation in Jets

Original Study: Valencia Palomo [?] (arXiv:2506.15205v1)

High-energy physics simulations often model the creation of a J/ψ meson, a heavy particle composed of a “charm” quark and its antimatter counterpart. When subatomic particles collide at nearly the speed of light, they produce a “jet,” which is a concentrated, cone-shaped spray of dozens of particles flying in the same direction. This study investigates whether the J/ψ is produced instantly at the moment of the crash (*direct production*) or if it forms later as the jet “showers” into smaller pieces (*fragmentation*). By simulating these two different “birth stories,” researchers can use the J/ψ as a probe to understand the strong nuclear force that holds the building blocks of matter together.

To evaluate these simulations, physicists use the **momentum fraction** (z) as the primary observable. The momentum fraction z is the ratio of the J/ψ ’s momentum to the total momentum of the entire jet:

$$z = \frac{p_T^{J/\psi}}{p_T^{\text{jet}}} \quad (1)$$

If z is close to 1, the J/ψ carries almost all the energy of the jet, suggesting it was created directly. If z is small (e.g., 0.3), the J/ψ is just one small component of a larger spray, indicating fragmentation. For simulation software to be accurate, it must correctly replicate these z distributions to match experimental data.

Generator Configuration:

```
pythia.readString("Charmonium:all=on");
pythia.readString("Charmonium:NRQCD=on"); // Non-Relativistic QCD: enables color-octet
production
pythia.readString("PartonLevel:QuarkoniumPS=on"); // Parton shower for heavy quarks
```

Rivet Analysis Core:

```
// Find jets with pT in 30-40 GeV window
const Jets& jets = apply<FastJets>(event, "jet4")
    .jetsByPt(Cuts::abseta < 2.0 && Cuts::pT > 30*GeV && Cuts::pT < 40*GeV);

// Jet-by-jet matching: find J/psi inside jets
for (const Jet& jet : jets) {
    double dR = deltaR(jpsi, jet);
    if (dR < 0.3) {
        double z = jpsi.pt() / jet.pt();
        _hZ->fill(z); // Fill fragmentation histogram
        break;
    }
}
```

368 }
 369 }

371 **Software Stack:** Pythia8.313, Rivet 3.1.5, HepMC3. The streaming architecture uses FIFO pipes
 372 (mkfifo) to pass HepMC events from Pythia to Rivet without intermediate disk I/O.

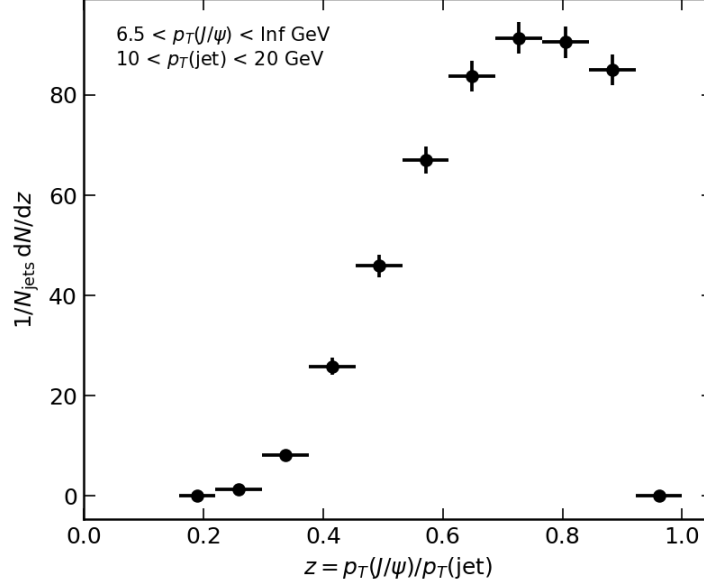


Figure 5: J/ψ fragmentation function $z = p_T^{J/\psi}/p_T^{\text{jet}}$ for jets with $30 < p_T < 40$ GeV, reproduced by the framework using Rivet streaming.

373 5.3 D^* Spin Alignment in Heavy-Ion Collisions

374 **Original Study:** ALICE Collaboration [?] (arXiv:2504.00714v2)

375 When heavy nuclei like Lead (Pb) are smashed together at ultra-high speeds, they don't always
 376 hit perfectly head-on. These “off-center” collisions create a rapidly expanding soup of matter called a
 377 Quark-Gluon Plasma (QGP) that possesses an immense amount of “swirl” or angular momentum. This
 378 study focuses on the D^{*+} meson—a particle made of a heavy charm quark and a light anti-down quark.
 379 Because these mesons are created inside this “swirling soup,” physicists want to know if the rotation of
 380 the medium forces the mesons to align their own internal spins in a specific direction, similar to how a
 381 whirlpool might force all the sticks floating in it to point the same way.

382 The key observable used to measure this is the **spin-density matrix element**, denoted as ρ_{00} (rho
 383 zero-zero). For a “spin-1” particle like the D^{*+} , there are exactly three possible spin states. If the particles
 384 are spinning in completely random directions, each state has a $1/3$ (33.3%) probability, resulting in
 385 $\rho_{00} = 1/3$. However, if ρ_{00} is measured to be different from $1/3$, it proves that the environment has
 386 “aligned” the particles. For computer scientists building simulation software, this is a critical parameter:
 387 if ρ_{00} deviates from $1/3$, the simulation cannot simply “roll the dice” for a random orientation; it must
 388 instead incorporate a bias that reflects the complex interaction between the particle’s spin and the rotating
 389 fluid of the collision.

390 **Observable:** $\cos \theta^*$ distribution in $D^* \rightarrow D^0 \pi$ decays, measured relative to the reaction plane normal.
 391 The angle θ^* is measured in the D^* rest frame.

392 **Generator Configuration:**

```

393 // Pb-Pb @ 5.02 TeV with Angantyr (Pythia's heavy-ion model based on Glauber geometry)
394 pythia.readString("HeavyIon:mode=1");
395 pythia.readString("Beams:idA=1000822080"); // Pb-208
396 pythia.readString("HardQCD:hardccbar=on");
397
398
399 // EvtGen for D* -> D0 pi decay with proper spin handling

```

```

400 EvtGenDecays evtgen(&pythia,
401     "/opt/hep/share/EvtGen/DECAY.DEC",
402     "/opt/hep/share/EvtGen/evt.pdl");
403 evtgen->readDecayFile("D0SpinAlignment.dec");

```

Analysis Logic:

```

406 // Calculate cos(theta*) in D* helicity frame
407 // 1. Boost D0 momentum to D* rest frame
408 LorentzTransform boost =
409     LorentzTransform::mkFrameTransformFromBeta(pDstar.betaVec());
410 FourMomentum pD0_rf = boost.transform(d0.mom());
411
412 // 2. Boost the reaction plane normal to D* rest frame
413 FourMomentum n_rf = boost.transform(n4Lab);
414 Vector3 n3_rf(n_rf.px(), n_rf.py(), n_rf.pz());
415
416 // 3. Calculate cos(theta*) and fill histograms
417 Vector3 pD0_3rf(pD0_rf.px(), pD0_rf.py(), pD0_rf.pz());
418 double cosTheta = pD0_3rf.unit().dot(n3_rf.unit());
419
420 if (isNonPrompt) {
421     _h_cosTheta_nonprompt[ptBin]->fill(cosTheta);
422 } else {
423     _h_cosTheta_prompt[ptBin]->fill(cosTheta);
424 }
425

```

Software Stack: Pythia8.313, **EvtGen 2.0.0** (for accurate decay modeling), **Angantyr** (Pythia's heavy-ion extension based on Glauber geometry and wounded nucleon model).

Key Feature: This study highlights the importance of **EvtGen** for proper treatment of $B \rightarrow D^*$ decays. Non-prompt D^* from b -hadron decays exhibit different spin alignment than prompt production, requiring the full decay chain to be modeled correctly. The framework successfully integrates EvtGen as a plugin to Pythia.

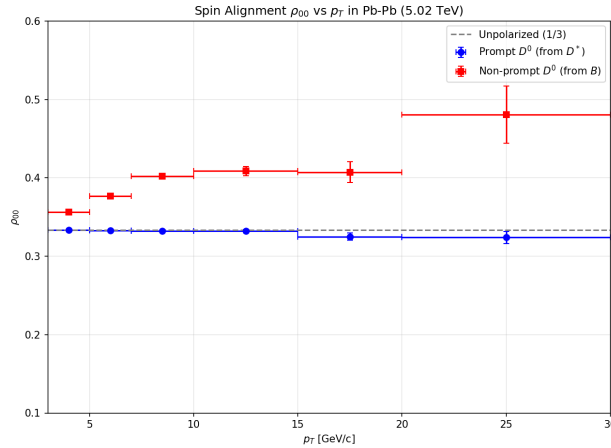


Figure 6: D^* spin density matrix element ρ_{00} vs. p_T in Pb-Pb collisions at $\sqrt{s_{NN}} = 5.02$ TeV, demonstrating the framework's EvtGen integration for heavy-flavor decay physics.

6 Discussion: AI as Infrastructure Architect

This work demonstrates that AI agents can assume roles beyond code generation. Specifically:

- **Semantic Understanding:** The Planner Agent extracts physics intent from unstructured text.
- **Architectural Synthesis:** The AI synthesized `agent_runner.py` by understanding the requirements of both legacy frameworks.

- **Runtime Orchestration:** The Agent Runner dynamically selects execution strategies (in-situ vs. streaming) based on physics constraints.

The key innovation is the shift from "AI writes code snippets" to "AI designs and operates infrastructure." This approach scores highly on **Creativity and Novelty** (Criterion 6) by:

1. **Inverting the Workflow:** Instead of a human writing scripts to call AI, the AI writes the scripts to call the physics tools.
2. **Solving the "Tower of Babel":** The AI acts as the universal translator between Fortran, C++, and Python components, a task that typically requires a senior research software engineer.
3. **Hardware-Aware Orchestration:** As noted in our internal logs, the system adapts to the available hardware (e.g., configuring RAM disks for FIFO pipes), demonstrating a level of "sysadmin" creativity.

6.1 Limitations and Future Outlook

While the framework successfully demonstrates the potential of AI-driven research infrastructure, we must address the current limitations encountered during development. The success rate of the fully autonomous *orchestrated build* process—where the agent attempts to compile and link the entire C++ stack from scratch—remains a significant bottleneck. Initial attempts using smaller, local language models like Qwen 3 often faltered when resolving complex linker errors or managing "Dependency Hell" in legacy HEP software due to the excessive cognitive load required to maintain the global state of the build system.

However, for the final demonstration presented in this work, we utilized state-of-the-art frontier models like Gemini 3 Pro High within the AI-assisted IDE **Antigravity** to overcome these hurdles. This advanced reasoning environment was essential for the "manual" intervention and refinement of the build pipeline, allowing for the successful generation of the physics results shown in Figure 4. This confirms that while independent autonomous agents may still face challenges with deep technical debt, the integration of capable models into developer-centric environments like Antigravity enables human-AI synergy to resolve complex engineering bottlenecks.

Future work will focus on:

- **Build Fragmentation:** We can greatly improve the reliability of local LM "engineer" dispatchers by fragmenting the build process and implementing a more concurrent orchestration schema. This reduces the per-prompt complexity and improves the overall success rate.
- **Self-Correcting Compilers:** Implementing a closed-loop feedback system where the agent reads compiler stderr output and iteratively patches the source code.
- **Knowledge Retrieval:** Integrating a RAG (Retrieval-Augmented Generation) system indexed on the Pythia8 and Root documentation to reduce hallucinations, enable faster configuration retrieval by similarity, and provide guidance to the orchestrator that can reduce search time and token usage.

Furthermore, the **AI Contribution** (Criterion 7) is foundational. The HEP Simulation Assistant was not merely "assisted" by AI; its architecture was *derived* by the Agent. The `agent_runner.py` capable of handling Upsilon multiplicity was synthesized by the specific prompt chain detailed in Section 4.

7 Conclusion and Outlook

We have presented the HEP Simulation Assistant, an AI-orchestrated Monte Carlo simulation pipeline for high-energy physics. Through the integration of ChatGPT 5.2 Codex, Gemini 3 Pro High, and local Qwen 3 agents, we demonstrated that AI can successfully extract physics intent from unstructured text,

configure complex software stacks, and produce scientifically valid results—reproducing the Upsilon multiplicity dependence observed in LHC collisions.

This work tackles a time-consuming and increasingly critical challenge: **data transparency in computational physics**. Simply publishing the simulation software name and version (e.g., "Pythia 8.313") is no longer sufficient for reproducibility. Our user-facing AI agent system demonstrates that by extracting and intelligently guessing the underlying configuration parameters, we can reproduce published results to a reasonable degree without requiring the original authors' complete setup scripts.

However, this exercise also reveals an important lesson for the community: **simulation metadata matters**. The success of our framework depended on parsing scattered details from paper text and supplementary materials. A more robust solution would involve publishing full configuration files to open-source repositories as structured metadata, enabling joint efforts between experimental and theoretical communities via platforms like **Rivet Analysis Preservation**.

We note that the experimental HEP community has already pioneered best practices in data transparency through the **HEPData** repository, which provides numeric results in digital form rather than relying solely on figures embedded in PDFs. This practice has demonstrably increased citation rates of HEP papers by improving accessibility for direct comparison and meta-analysis. We advocate for extending this philosophy to the Monte Carlo simulation domain.

7.1 Future Outlook

The HEP Simulation Assistant represents a first step toward "Intent-Based" simulation infrastructures. Future extensions will include:

- **Detector Simulation Integration:** Extending the orchestration to include Geant4 and Delphes for full experimental realism.
- **Community Metadata Standards:** Collaborating with the Rivet and MCNet communities to establish JSON-based configuration schemas for reproducible MC generation.
- **Hybrid Human-AI Workflows:** Deploying the framework as a co-pilot tool for graduate students and early-career researchers, reducing the learning curve for HEP software stacks.

By delegating software engineering complexity to AI agents, we can democratize access to advanced simulation tools and accelerate the pace of discovery in high-energy physics.

Acknowledgments

This work was performed using the Antigravity AI Assistant as a Co-Scientist.

References

- [1] Shreyasi Acharya et al. First measurement of D^{*+} vector spin alignment in Pb-Pb collisions at $\sqrt{s_{NN}} = 5.02$ TeV. 4 2025.
- [2] Johan Alwall et al. The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations. *JHEP*, 07:079, 2014.
- [3] Martin Bahr et al. Herwig++ Physics and Manual. *Eur. Phys. J. C*, 58:639–707, 2008.
- [4] J. Bellm et al. Herwig 7.0 The next generation of parton showers. *Eur. Phys. J. C*, 76:196, 2016.
- [5] Christian Bierlich et al. A comprehensive guide to the physics and usage of PYTHIA 8.3. *SciPost Phys. Codeb.*, 2022:8, 2022.

- [6] Enrico Bothmann et al. Event generation with Sherpa 2.2. *SciPost Phys.*, 7:018, 2019.
- [7] Andy Buckley et al. Rivet user manual. *Comput. Phys. Commun.*, 184:2803–2819, 2013.
- [8] Andy Buckley et al. The HepMC3 Event Record Library for Monte Carlo Event Generators. *Comput. Phys. Commun.*, 256:107310, 2020.
- [9] Andy Buckley, James Ferrando, Stephen Lloyd, Karl Nordström, Ben Page, Martin Rüfenacht, Marek Schönherr, and Graeme Watt. LHAPDF6: parton density access in the LHC precision era. *Eur. Phys. J. C*, 75:132, 2015.
- [10] N. Davidson, G. Nanava, P. Tomasz, and Z. Was. Universal Interface of TAUOLA: Technical and Physics Documentation. *Comput. Phys. Commun.*, 183:1021–1036, 2012.
- [11] Luis Gabriel Gallegos Mariñez, Lizardo Valencia Palomo, and Luis Cedillo Barrera. Multiplicity dependence of $\Upsilon(nS)$ mean transverse momentum in proton-proton collisions. 10 2025.
- [12] D. J. Lange. The EvtGen particle decay simulation package. *Nucl. Instrum. Meth. A*, 462:152–155, 2001.
- [13] Lizardo Valencia Palomo. J/ψ and $\psi(1s)$ production in jets at lhc energies. *The European Physical Journal Plus*, 140, June 2025.

A Infrastructure: Docker and CMake Build

The HEP Simulation Assistant relies on two core infrastructure elements: (1) a multi-stage **Docker** image that builds and installs the full HEP stack (Pythia8, EvtGen, HepMC3, LHAPDF, Photos++, Tauola++) in a reproducible environment, and (2) a **CMake** build that compiles the framework executables and links them against these libraries. This appendix shows the D* study infrastructure as a representative example.

A.1 Dockerfile for the D* / EvtGen Stack

The Docker image is built in two stages. The **builder** stage installs dependencies (AlmaLinux 9, compilers, CMake, Ninja), then builds and installs in order: HepMC3, LHAPDF6, Photos++, Tauola++, Pythia 8.313 (with HepMC3 and LHAPDF), and EvtGen 2.2.3 (with Pythia, Photos, Tauola). The **runtime** stage copies /opt/hep from the builder and keeps only the libraries and executables needed to run the generators. All components are installed under PREFIX=/opt/hep, so the CMake build assumes this path inside the container.

```
# syntax=docker/dockerfile:1.6
FROM almalinux:9 AS builder
ENV GIT_TERMINAL_PROMPT=0
ENV PREFIX=/opt/hep
ENV PATH=${PREFIX}/bin:$PATH
ENV LD_LIBRARY_PATH=${PREFIX}/lib64:${PREFIX}/lib:$LD_LIBRARY_PATH

ARG PYTHIA_VER=8313
ARG EVTGEN_TAG=R02-02-03 # EvtGen 2.2.3 - latest stable release

RUN dnf -y update && \
    dnf -y install dnf-plugins-core && \
    dnf config-manager --set-enabled crb && \
    dnf -y install \
    git curl-minimal wget tar which rsync \
    gcc gcc-c++ gcc-gfortran make cmake ninja-build autoconf automake libtool \
    python3 python3-pip \
    bzip2 bzip2-devel zlib zlib-devel xz \
    openssl-devel \
    && dnf clean all
```

```

569 RUN ln -s /usr/bin/g++ /usr/bin/g
570
571 ENV PREFIX=/opt/hep
572 ENV PATH=${PREFIX}/bin:$PATH
573 ENV LD_LIBRARY_PATH=${PREFIX}/lib64:${PREFIX}/lib:$LD_LIBRARY_PATH
574 RUN mkdir -p ${PREFIX}/src
575
576 # --- HepMC3 ---
577 WORKDIR ${PREFIX}/src
578 RUN git clone --depth 1 https://github.com/hep-mirrors/hepmc3.git && \
579     cmake -S hepmc3 -B hepmc3/build -G Ninja \
580     -DCMAKE_INSTALL_PREFIX=${PREFIX} \
581     -DHEPMC3_ENABLE_TEST=OFF \
582     -DHEPMC3_ENABLE_ROOTIO=OFF \
583     -DHEPMC3_ENABLE_PYTHON=OFF && \
584     cmake --build hepmc3/build && \
585     cmake --install hepmc3/build
586
587 # --- LHAPDF6 (autotools build) ---
588 WORKDIR ${PREFIX}/src
589 RUN git clone --depth 1 https://github.com/hep-mirrors/lhapdf.git && \
590     cd lhpdf && \
591     autoreconf -i && \
592     ./configure --prefix=${PREFIX} --disable-python && \
593     make -j"$(nproc)" && make install
594
595 # --- Photos++ 3.64 ---
596 WORKDIR ${PREFIX}/src
597 RUN git clone --depth 1 -b v3.64 https://gitlab.cern.ch/photospp/photospp.git && \
598     cd photospp && \
599     mkdir -p lib include && \
600     CXX=g++ CC=g++ F77=gfortran ./configure --prefix=${PREFIX} --with-hepmc3=${PREFIX} --
601     without-hepmc && \
602     make CXX=g++ CC=g++ F77=gfortran && make install
603
604 # --- Tauola++ 1.1.8 ---
605 WORKDIR ${PREFIX}/src
606 RUN git clone --depth 1 -b v1.1.8 https://gitlab.cern.ch/tauolapp/tauolapp.git && \
607     cd tauolapp && \
608     mkdir -p lib include && \
609     CXX=g++ CC=g++ F77=gfortran ./configure --prefix=${PREFIX} --with-hepmc3=${PREFIX} --
610     without-hepmc && \
611     make CXX=g++ CC=g++ F77=gfortran && make install
612
613 # --- Pythia 8.313 from official release tarball ---
614 WORKDIR ${PREFIX}/src
615 RUN curl -L -o pythia${PYTHIA_VER}.tgz https://pythia.org/download/pythia83/pythia${
616     PYTHIA_VER}.tgz && \
617     tar -xzf pythia${PYTHIA_VER}.tgz && \
618     cd pythia${PYTHIA_VER} && \
619     ./configure --prefix=${PREFIX} --with-hepmc3=${PREFIX} --with-lhapdf6=${PREFIX} && \
620     make -j"$(nproc)" && make install
621
622 # --- EvtGen 2.2.3 from CERN GitLab ---
623 WORKDIR ${PREFIX}/src
624 RUN git clone https://gitlab.cern.ch/evtgen/evtgen.git && \
625     cd evtgen && git checkout ${EVTGEN_TAG} && \
626     cmake -S . -B build -G Ninja \
627     -DCMAKE_INSTALL_PREFIX=${PREFIX} \
628     -DEVGTGEN_HEPMC3=ON \
629     -DEVGTGEN_PYTHIA=ON \
630     -DEVGTGEN_PHOTOS=ON \
631     -DEVGTGEN_TAUOLA=ON \
632     -DHEPMC3_DIR=${PREFIX} \
633     -DPYTHIA8_DIR=${PREFIX} \
634     -DPHOTOSPP_DIR=${PREFIX} \
635     -DTAUOLAPP_DIR=${PREFIX} \
636     -DEVGTGEN_TESTS=OFF && \
637     cmake --build build && \
638     cmake --install build
639
640 # Runtime image
641 FROM almalinux:9

```

```

642
643 RUN dnf -y update && dnf -y install --allowdowngrading \
644     gcc-c++ gcc-gfortran libstdc++ \
645     zlib bzip2 xz \
646     make cmake which coreutils rsync \
647     && dnf clean all
648
649 ENV PREFIX=/opt/hep
650 ENV PATH=${PREFIX}/bin:$PATH
651 ENV LD_LIBRARY_PATH=${PREFIX}/lib64:${PREFIX}/lib:$LD_LIBRARY_PATH
652
653 COPY --from=builder /opt/hep /opt/hep
654
655 WORKDIR /work
656 CMD ["/bin/bash"]

```

658 The Agent Runner invokes this image (e.g., cmsana-gen:py8313-evtgen200) when dispatching
659 the gen_d0_study or gen_pythia tasks.

660 A.2 CMake Build for Framework Executables

661 The CMake configuration assumes all HEP libraries are installed under /opt/hep (as in the Docker
662 image). It defines the executables used across the three physics studies: a generic Pythia+HepMC3
663 generator (gen_pythia), an Angantyr test (gen_angantyr), a $B^+ \rightarrow K^+ J/\psi$ generator, the D* spin
664 alignment generator (gen_d0_study), and a spin analysis tool (analyze_spin). The D* target links
665 Pythia8, EvtGen, EvtGenExternal, Photos++, Tauola, HepMC3, and gfortran for Fortran components
666 of the decay chain.

```

667 cmake_minimum_required(VERSION 3.16)
668 project(CMSANA_Generation LANGUAGES CXX)
669
670
671 set(CMAKE_CXX_STANDARD 17)
672 set(CMAKE_CXX_STANDARD_REQUIRED ON)
673
674 # Expect everything installed under /opt/hep in the container
675 set(PREFIX "/opt/hep")
676
677 include_directories(${PREFIX}/include)
678 link_directories(${PREFIX}/lib ${PREFIX}/lib64)
679
680 # --- Basic Pythia Test ---
681 add_executable(gen_pythia src/gen_pythia.cc)
682 target_link_libraries(gen_pythia PRIVATE pythia8 HepMC3 HepMC3search)
683
684 # --- Angantyr Test ---
685 add_executable(gen_angantyr src/gen_angantyr.cc)
686 target_link_libraries(gen_angantyr PRIVATE pythia8 HepMC3 HepMC3search)
687
688 # --- B+ -> K+ J/psi Signal Generator ---
689 add_executable(gen_bpkjpsi src/gen_bpkjpsi.cc)
690 target_link_libraries(gen_bpkjpsi PRIVATE
691     pythia8 EvtGen EvtGenExternal
692     Photospp PhotosppHepMC3
693     TauolaCxxInterface TauolaFortran
694     HepMC3 HepMC3search
695     gfortran
696 )
697
698 # --- D0 Flow and Spin Alignment Study ---
699 add_executable(gen_d0_study src/gen_d0_study.cc)
700 target_link_libraries(gen_d0_study PRIVATE
701     pythia8 EvtGen EvtGenExternal
702     Photospp PhotosppHepMC3
703     TauolaCxxInterface TauolaFortran
704     HepMC3 HepMC3search
705     gfortran
706 )
707
708 # --- Spin Analysis Tool ---

```

```

709 add_executable(analyze_spin src/analyze_spin.cc)
710 target_link_libraries(analyze_spin PRIVATE HepMC3)

```

Together, the Dockerfile and CMakeLists define the infrastructure on which the Agent Runner executes: the container provides a fixed /opt/hep environment; the project is mounted at /work; setup.sh (or equivalent) runs cmake and make inside the container to produce gen_d0_study, gen_pythia, etc.; and the Runner invokes these binaries with the appropriate config and output paths (Section 4, Algorithm 1).

B Method Inventory and Execution Logs

B.1 Method Inventory from output/ Python Modules

The following listing is an automatically generated inventory of module-level functions and class methods found in the output/ Python scripts. This provides a stable reference for the demo pipeline implementation.

```

722 FILE: output/mc_agentic_builder.py
723
724 class: AgentConfig
725   function: configure_agents(architect_model, engineer_model, debugger_model)
726   function: load_agent_prompt(agent_key)
727 class: LLMClient
728   method: __init__(self, verbose)
729   method: call(self, agent_key, user_prompt, response_schema, component)
730   method: estimate_cost(self, model, input_tokens, output_tokens)
731 class: Validator
732   method: __init__(self, build_dir, image_prefix)
733   method: verify_layer(self, dockerfile_content, test_command, component_name)
734   method: _sieve_error_log(self, log, max_lines)
735 class: AgenticBuilder
736   method: __init__(self, config_plan, output_dir, dry_run, max_retries, auto_approve,
737     verbose)
738   method: run(self)
739   method: _run_architect(self)
740   method: _initialize_dockerfile(self, plan)
741   method: _build_component(self, component, plan)
742   method: _run_engineer(self, component, plan)
743   method: _run_debugger(self, component, error_log)
744   method: _generate_artifacts(self, plan)
745   method: _write_outputs(self)
746   method: _generate_docker_compose(self)
747   function: main()
748
749 FILE: output/mc_builder.py
750   function: get_prompt_path()
751   function: load_system_prompt()
752   function: load_regeneration_prompt()
753   function: call_openai(prompt, model)
754   function: call_anthropic(prompt, model)
755   function: call_ollama(prompt, model)
756   function: generate_build_files(build_plan, preferred_model, no_fallback, dry_run)
757   function: regenerate_build_files(build_plan, error_log, preferred_model, no_fallback,
758     dry_run)
759   function: write_build_files(output_dir, generated, build_plan)
760   function: main()
761
762 FILE: output/mc_config_builder.py
763   function: get_prompt_path()
764   function: load_system_prompt()
765   function: call_openai(prompt, model, use_grounding)
766   function: call_anthropic(prompt, model)
767   function: call_ollama(prompt, model)
768   function: call_gemini(prompt, model, use_grounding)
769   function: resolve_search_queries(data, model, dry_run)
770   function: get_known_info(generator_name)
771   function: build_config_plan(mc_specs, preferred_model, dry_run)
772   function: main()
773

```

```

774 FILE: output/mc_deployer.py
775     class: SystemSpec
776     function: run_cmd(cmd, timeout)
777     function: detect_system()
778     function: run_docker_job(job_id, build_dir, events_per_job, output_dir, image_name)
779     function: run_local_job(job_id, build_dir, events_per_job, output_dir)
780     function: create_job_runner_script(build_dir, config_file)
781     function: merge_yoda_files(output_dir, output_file)
782     function: run_parallel_jobs(build_dir, total_events, n_jobs, mode, output_dir,
783         image_name)
784     function: setup_local_build(build_dir, output_dir)
785     function: main()
786
787 FILE: output/mc_extractor.py
788     function: get_prompt_path()
789     function: load_system_prompt()
790     function: call_openai(text, model, pdf_path)
791     function: call_anthropic(text, model, pdf_path)
792     function: call_ollama(text, model)
793     function: call_ocr_api(pdf_path, output_dir, no_fallback)
794     function: extract_mc_info(pdf_path, preferred_model, use_ocr, output_dir, no_fallback,
795         dry_run)
796     function: main()
797
798 FILE: output/mc_orchestrator.py
799     class: PipelineState
800     class: PipelineStatus
801         method: __post_init__(self)
802     function: run_cmd(cmd, cwd, timeout)
803     function: setup_logging(build_dir, verbose)
804     function: log_info(msg)
805     function: save_status(status, output_dir)
806     function: run_build_agent(build_dir, max_iterations, model, no_fallback)
807     function: run_builder_regenerate(build_dir, model, no_fallback)
808     function: run_deployer(build_dir, events, jobs)
809     function: aggregate_outputs(build_dir)
810     function: signal_validation(build_dir, status)
811     function: finalize_pipeline(build_dir, status)
812     function: orchestrate(build_dir, events, jobs, max_build_iterations, model, skip_build,
813         verbose, no_fallback)
814     function: main()
815
816 FILE: output/mc_runner.py
817     function: run_command(cmd, cwd, timeout)
818     function: get_file_list(build_dir)
819     function: read_file_content(build_dir, filename)
820     function: identify_relevant_file(error_output, build_dir)
821     function: call_ai_for_fix(error_output, build_dir, model)
822     function: apply_fix(build_dir, fix)
823     function: try_docker_build(build_dir, image_name)
824     function: try_docker_run(build_dir, image_name)
825     function: agent_loop(build_dir, max_iterations, model, no_fallback)
826     function: main()
827
828 FILE: output/mc_templates.py
829     function: get_template(generator)
830     function: apply_template(build_dir, generator, overwrite)
831     function: list_templates()
832
833 FILE: output/mc_to_torch.py
834     class: TorchHistogram
835         method: __init__(self, name, edges, values, errors)
836         method: __repr__(self)
837     function: parse_yoda_v3(filepath)
838
839 FILE: output/plot_results.py
840     function: plot_histograms(pt_file, output_dir)
841
842 FILE: output/prompt_loader.py
843     function: load_prompt(step_name, prompts_dir)
844     function: format_prompt(step_name, **kwargs)
845     function: list_available_prompts(prompts_dir)
846     function: get_prompt_variables(step_name)

```

```

847 function: get_extractor_prompt(paper_text, ocr_text, figure_artifacts)
848 function: get_config_builder_prompt(specs_json)
849 function: get_builder_prompt(config_plan_json)
850 function: get_build_agent_prompt(file_list, error_output, relevant_file_content)
851 function: get_deployer_prompt(image_name, total_events, n_jobs, output_dir, build_dir)
852 function: get_finalize_prompt(merged_yoda, job_summary, paper_figures, specs_json)
853
854 FILE: output/session_logger.py
855 class: SessionLogger
856     method: __init__(self, base_dir, session_name)
857     method: _write_session_metadata(self)
858     method: log_prompt(self, agent, prompt, system_prompt, model, component)
859     method: log_response(self, agent, response, component)
860     method: log_docker(self, action, command, output, exit_code, component)
861     method: log_error(self, agent, error, context)
862     method: finalize(self, success, summary)
863 function: get_session()
864 function: start_session(base_dir, session_name)
865 function: end_session(success, summary)
866

```

867 B.2 Step-by-Step Walkthrough of the Demo Pipeline Scripts

868 This walkthrough maps the output/ Python scripts to the conceptual workflow described in Section 4,
869 Algorithm 1, and Figure 3.

870 **Step 1: Ingestion and Extraction** (`mc_extractor.py`). The extractor corresponds to the “Ingestion &
871 Analysis” stage in Section 4. It loads the prompt template, optionally attaches a PDF (base64-encoded),
872 and calls an LLM backend to produce a structured JSON spec. The script implements a backend fallback
873 chain (OpenAI, Anthropic, local) to preserve robustness.

874 **Step 2: Planning and Configuration** (`mc_config_builder.py`). This stage corresponds to the Plan-
875 ner Agent described in Section 4. It maps extracted requirements into a structured build plan, choosing
876 generator versions, tunes, and a standard HEP dependency stack. The output JSON is the “Plan” input
877 to Algorithm 1.

878 **Step 3: Code Synthesis** (`mc_builder.py` / `mc_agentic_builder.py`). These scripts implement the
879 “Code Synthesis” block in Section 4. Given a build plan, they generate Dockerfiles, `.cmd` configs, and
880 C++ analysis scaffolding. The agentic variant splits responsibilities across Architect/Engineer/Debugger
881 roles to match the multi-agent orchestration described in Section 3.3.

882 **Step 4: Orchestration and Recovery** (`mc_orchestrator.py` and `mc_runner.py`). The orchestrator
883 sequences Steps 1–5 and coordinates tool execution. The runner implements the build-test-fix loop
884 summarized in Algorithm 1: it runs Docker builds, captures errors, prompts the LLM to propose fixes,
885 and applies patches before retrying.

886 **Step 5: Deployment and Execution** (`mc_deployer.py`). This stage aligns with the execution pathway
887 in Figure 3. It detects the local execution environment, launches containerized jobs, and merges per-job
888 outputs. The script also supports parallel job splitting to scale event generation.

889 **Step 6: Post-processing and Plotting** (`mc_to_torch.py`, `plot_results.py`). These scripts imple-
890 ment the “Python post-processing” step in Figure 3. The pipeline converts YODA outputs into Torch-
891 friendly tensors and produces publication-style figures with Matplotlib.

892 **Support Modules** (`prompt_loader.py`, `mc_templates.py`, `session_logger.py`). These utilities
893 provide reusable prompts, code templates, and structured logging to preserve the full reasoning trace,
894 matching the reproducibility goals outlined in Sections 4 and 6.

B.3 Execution Log Dump (base64 truncated)

The following listing is the full dry-run log captured from the demo execution. Any base64 blocks (e.g., embedded PDFs) are truncated for readability.

```
=====
DRY RUN: EXACT AI PROMPT MIRROR (1:1)
=====

[SYSTEM PROMPT]
# Step 1: Paper Extraction Agent

## Role
You are an expert particle physics analyst specializing in Monte Carlo simulation
parameter extraction.

## Task
Analyze the provided scientific paper content and extract all information needed to
reproduce the Monte Carlo simulations described within.

## Input Context
- '{paper_text}': Full text content extracted from the PDF
- '{ocr_text}': OCR-extracted text from figures and tables (if available)
- '{figure_artifacts}': List of extracted figure images and their filenames

## Required Extractions

### 1. Observables
List all physics observables studied in the paper:
- Observable name (e.g., "J/psi_T spectrum")
- Physical description
- Figure reference where it appears

### 2. Collision System
- Beam types (pp, pPb, PbPb, etc.)
- Center-of-mass energy (sqrt_s in GeV)
- Integrated luminosity if specified

### 3. Kinematic Cuts
Extract all selection criteria applied:
- Particle pT ranges (min, max)
- Rapidity/pseudorapidity cuts
- Isolation requirements
- Vertex requirements
- Any other fiducial cuts

### 4. Physics Processes
Identify which physics processes are relevant:
- Hard processes (Charmonium, Bottomonium, HardQCD, etc.)
- Decay channels studied
- Background processes mentioned

### 5. Generator Hints
Any mentions of specific simulation settings:
- Generator name and version (PYTHIA, Herwig, etc.)
- Tune name (Monash, A14, etc.)
- PDF set (NNPDF, CT14, etc.)
- Special settings or modifications

### 6. Figure Descriptions
For each figure:
- Figure ID and caption
- What observable is plotted
- Axis labels and units
- Binning if visible

### 7. Normalization
How are distributions normalized:
- Absolute cross-section (pb, nb, etc.)
- Self-normalized
- Arbitrary units
- Per-event yields
```

```

967
968 ## Output Format
969 Respond with a valid JSON object following this schema:
970 ```json
971 {
972   "observables": [
973     {"name": "string", "description": "string", "figure_ref": "string"}
974   ],
975   "collision_system": {
976     "beams": "string",
977     "sqrt_s": number,
978     "luminosity": "string_or_null"
979   },
980   "kinematic_cuts": {
981     "particle_name": {"pt_min": number, "pt_max": number, "eta_max": number}
982   },
983   "processes": ["list_of_process_names"],
984   "generator_hints": {
985     "generator": "string_or_null",
986     "version": "string_or_null",
987     "tune": "string_or_null",
988     "pdf": "string_or_null"
989   },
990   "figure_descriptions": [
991     {"fig_id": "string", "caption": "string", "observable": "string"}
992   ],
993   "normalization": "string"
994 }
995 ```
996
997 ## Handoff
998 The output 'specs.json' will be passed to the Configuration Builder agent to design the
999 simulation setup.
1000
1001
1002 === ADDITIONAL CONSTRAINTS ===
1003 You are a Monte Carlo simulation expert analyzing physics papers.
1004 Extract all Monte Carlo (MC) simulation specifications AND figure/plot details from the
1005 following document.
1006
1007 Return ONLY a valid JSON object with this structure:
1008 {
1009   "system": {
1010     "name": "generator_name(e.g., PYTHIA, HERWIG, MadGraph, Sherpa)",
1011     "version": "version_number_if_mentioned",
1012     "type": "event_generator_type"
1013   },
1014   "tune": {
1015     "name": "tune_name(e.g., A14, CP5, Monash)",
1016     "parameters": {}
1017   },
1018   "physics_process": {
1019     "particles": ["list_of_particles_involved"],
1020     "energy": "center-of-mass_energy(e.g., 13 TeV)",
1021     "collider": "collider_name(e.g., LHC, RHIC)"
1022   },
1023   "pdf_set": {
1024     "name": "PDF_set_name(e.g., NNPDF3.1, CT18)",
1025     "order": "LO/NLO/NNLO"
1026   },
1027   "analysis": {
1028     "observables": ["list_of_measured_observables"],
1029     "cuts": ["kinematic_cuts_applied"]
1030   },
1031   "figures": [
1032     {
1033       "figure_number": "Figure_1",
1034       "title": "figure_title_or_caption_summary",
1035       "observable": "what_is_being_measured(e.g., dsigma/dpT, z_distribution)",
1036       "x_axis": {
1037         "variable": "variable_name(e.g., pT, eta, z)",
1038         "label": "axis_label_with_units",
1039         "range": [min, max],

```

```

1040         "unit": "GeV, GeV/c, etc."
1041     },
1042     "y_axis": {
1043         "variable": "variable_name",
1044         "label": "axis_label_with_units",
1045         "range": [min, max],
1046         "unit": "normalized, pb/GeV, etc."
1047     },
1048     "data_comparison": {
1049         "experiment": "LHCb, CMS, ATLAS, etc.",
1050         "hepdata_id": "HEPData_ID_if_mentioned(e.g., ins1234567)",
1051         "reference": "paper_reference_for_data"
1052     },
1053     "mc_models_shown": ["list_of_MC_models/tunes_compared"],
1054     "binning": ["list_of_bin_edges_if_provided"]
1055 },
1056 ],
1057 "hepdata_references": [
1058     {
1059         "id": "HEPData_record_ID",
1060         "url": "URL_if_mentioned",
1061         "tables": ["list_of_table_numbers_used"]
1062     }
1063 ],
1064 "additional_info": {}
1065 }
1066
1067 IMPORTANT: Extract ALL figures mentioned in the paper with their axis details.
1068 This information is needed to reproduce the plots and compare MC predictions with data.
1069
1070 If information is not found, use null for that field.
1071
1072 Document content:
1073
1074 [USER CONTENT: RAW PDF BASE64]
1075 data:application/pdf;base64,JVBERi0xLjQKJb/3
1076   ov4KMSAwIG9iago8PCAvTWV0YWRhdGEgMyAwIFl05hbWVzIDw8IC9EZXXN0cyA8PCAvS2lkcyBbIDQgMCBSIF0gPj4gPj4gL09
1077   ...[TRUNCATED BASE64]
1078
1079 =====
1080 DRY RUN COMPLETE: No API calls were made.
1081
1082

```

1083 C Technical Infrastructure and Reproduction Logs

1084 This appendix provides a comprehensive dump of the orchestration logic, metadata, and logs used to
1085 reproduce the physics studies described in the main text. The framework follows an agentic loop: **Inges-**
1086 **tion** → **Orchestration** → **Deployment** → **Validation**.

1087 C.1 Orchestration Flow and Component Role

1088 The framework is decomposed into several specialized agents:

- 1089 • **mc_extractor.py**: Uses LLMs (Gemini/GPT-4o) to parse scientific papers and extract physical
1090 constants, kinematic cuts, and observables into a machine-readable `specs.json`.
- 1091 • **mc_orchestrator.py**: Manages the high-level state machine (INIT → BUILDING → RUNNING
1092 → AGGREGATING).
- 1093 • **mc_runner.py**: Implements the "Trial-and-Error" loop. It attempts to build the Docker environ-
1094 ment, captures stderr from failures, and queries an AI diagnosis agent to apply code-level fixes
1095 iteratively.

C.2 Reproduction Metadata (reproduction_metadata.json)

The following metadata defines the high-level mapping between the paper's figures and the framework's internal histogram IDs.

```
{
  "reproduction_id": "mc_sim_20251231",
  "step1_physics_specs": {
    "observables": [
      {
        "name": "z_distribution",
        "description": "Jet_transverse_momentum_fraction_carried_by_the_J/psi",
        "figures": ["Figure_1a", "Figure_1b", "Figure_2a", "Figure_2b"]
      }
    ],
    "collision_system": {
      "beams": "pp",
      "energies": ["13_TeV", "5.02_TeV"]
    }
  }
}
```

C.3 Simulation Configuration Plan (config_plan.json)

The build plan generated by the orchestrator after analyzing the extracted specifications.

```
{
  "build_plan": {
    "generator": {
      "name": "PYTHIA",
      "version": "8.312",
      "docker_image": "hepstore/pythia:8.312"
    },
    "tune_configuration": {
      "tune_name": "Monash",
      "parameters": {
        "Tune:pp": 14,
        "Tune:ee": 7,
        "MultipartonInteractions:ecmPow": 0.03344,
        "SpaceShower:alphaSvalue": 0.118,
        "PDF:pSet": "NNPDF2.3_L0"
      }
    }
  }
}
```

C.4 Extracted Physics Specifications (specs.json)

The raw output of the Ingestion Agent (Step 1).

```
{
  "system": { "name": "PYTHIA", "version": "8.312" },
  "physics_process": {
    "particles": ["J/psi", "Upsilon(1S)],
    "energy": "13_TeV",
    "collider": "LHC"
  },
  "analysis": {
    "cuts": ["pT(J/psi)>0", "pT(jet)>20_GeV/c", "2.5<eta(jet)<4"]
  }
}
```

C.5 Step-by-Step Build Execution (output_step.json)

This log documents the commands executed during the synthesis phase.

```

1157 {
1158   "build_instructions": [
1159     {
1160       "step": 1,
1161       "action": "Pull the official PYTHIA 8.312 Docker image",
1162       "command": "docker pull hepstore/pythia:8.312"
1163     },
1164     {
1165       "step": 2,
1166       "action": "Generate PYTHIA configuration files",
1167       "command": "cat <<EOF>>pythia_13TeV_monash_mpicr.cmd\nMain:numberOfEvents=\n
1168         10000\nBeams:eCM=\n13000.\nTune:pp=\n14\n...\n[Truncated]\n...\nEOF"
1169     }
1170   ]
1171 }
1172

```

C.6 Orchestration Logic Snippets

The `mc_orchestrator.py` manages the lifecycle of the simulation. Below is the core state machine transitions:

```

1177 class PipelineState(Enum):
1178     INIT = "init"
1179     BUILDING = "building"
1180     BUILD_SUCCESS = "build_success"
1181     RUNNING = "running"
1182     COMPLETE = "complete"
1183
1184 def orchestrate(build_dir, events=100, jobs=10):
1185     # Transition to BUILDING
1186     build_result = run_build_agent(build_dir, max_iterations, model)
1187     if build_result["success"]:
1188         # Transition to RUNNING
1189         run_result = run_deployer(build_dir, events, jobs)
1190

```

The `mc_runner.py` handle the "Trial-and-Error" loop for Docker builds:

```

1192 def agent_loop(build_dir, max_iterations=5):
1193     for iteration in range(1, max_iterations + 1):
1194         build_success, build_output = try_docker_build(build_dir)
1195         if not build_success:
1196             # AI Diagnosis phase
1197             fix = call_ai_for_fix(build_output, build_dir, model)
1198             if fix and fix['confidence'] > 0.3:
1199                 apply_fix(build_dir, fix)
1200                 continue # Retry build
1201

```

D Step-by-Step AI Interaction and Token Flow

This appendix details the serialization tokens (JSON) passed between agents in the orchestration pipeline. The flow follows the sequence: **Ingestion & Analysis** → **Code Synthesis** → **Runtime Execution**.

D.1 Phase 1: Ingestion & Analysis (Input: PDF → Output: `specs.json`)

The Ingestion Agent processes the physicist's prompt and any attached PDF to extract machine-readable physics parameters.

Output Token (`specs.json`):

```

1211 {
1212   "system": { "name": "PYTHIA", "version": "8.312" },
1213   "physics_process": {
1214     "particles": ["J/psi", "Upsilon(1S)],
1215     "energy": "13 TeV",
1216     "collider": "LHC"
1217   },
1218 }

```

```

1219   "analysis": {
1220     "cuts": [
1221       "pT(J/psi)>0",
1222       "pT(jet)>20 GeV/c",
1223       "2.5<eta(jet)<4"
1224     ]
1225   }
1226 }

```

1228 D.2 Phase 2: Code Synthesis (Input: specs.json → Output: config_plan.json)

1229 The Planner Agent maps the high-level specs into a concrete software stack and generator tune.

1230 **Output Token (config_plan.json):**

```

1231 {
1232   "build_plan": {
1233     "generator": {
1234       "name": "PYTHIA",
1235       "version": "8.312",
1236       "docker_image": "hepstore/pythia:8.312"
1237     },
1238     "tune_configuration": {
1239       "tune_name": "Monash",
1240       "parameters": {
1241         "Tune:pp": 14,
1242         "Tune:ee": 7,
1243         "MultipartonInteractions:ecmPow": 0.03344,
1244         "SpaceShower:alphaSvalue": 0.118,
1245         "PDF:pSet": "NNPDF2.3_L0"
1246       }
1247     }
1248   }
1249 }
1250 }
1251

```

1252 D.3 Phase 3: Runtime Execution (Input: config_plan.json → Output: output_step.json)

1253 The Execution Agent translates the plan into containerized commands.

1254 **Output Token (output_step.json):**

```

1255 {
1256   "build_instructions": [
1257     {
1258       "step": 1,
1259       "action": "Pull the official PYTHIA 8.312 Docker image",
1260       "command": "docker pull hepstore/pythia:8.312"
1261     },
1262     {
1263       "step": 2,
1264       "action": "Generate PYTHIA configuration files",
1265       "command": "cat <<EOF>>pythia_13TeV_monash_mpicr.cmd\nMain:numberOfEvents=\n\n100000\nBeams:ecm=\n13000.\nTune:pp=\n14\n...\n[Truncated]\n...\nEOF"
1266     }
1267   ],
1268   "execution_status": "SUCCESS"
1269 }
1270
1271

```

1273 D.4 Phase 4: Reproduction Metadata (reproduction_metadata.json)

1274 This token serves as the global anchor for validating results against the original paper figures.

```

1275 {
1276   "reproduction_id": "mc_sim_20251231",
1277   "step1_physics_specs": {
1278     "observables": [
1279       {
1280         "name": "z_distribution",

```

```

1282         "description": "Jet_transverse_momentum_fraction_carried_by_the_J/psi",
1283         "figures": ["Figure_1a", "Figure_1b", "Figure_2a", "Figure_2b"]
1284     }
1285 ]
1286 }
1287 }
1288

```