

1      **Autonomous Orchestration of Open-Source Monte Carlo**  
2      **Pipelines via Agentic AI:**

3      **A Novel Approach to Research Infrastructure in High-Energy**  
4      **Physics**

5      Anonymous Author

6      *Department of Physics, Anonymous University*

7      *Anonymous City, Anonymous Country*

8      anonymous@email.edu

9      January 2026

10     **Abstract**

11    Monte Carlo (MC) event generation is a foundational computational technique in high-  
12    energy physics (HEP), enabling researchers to simulate collisions between fundamental particles and  
13    predict experimental outcomes. However, the complexity of MC pipelines—requiring deep integration  
14    of quantum field theory, specialized software libraries, and high-performance computing—has  
15    traditionally made this domain inaccessible to automation. In this work, we present a novel methodology  
16    utilizing Large Language Models (LLMs) as autonomous *Research Software Engineers* to  
17    synthesize, configure, and orchestrate MC simulation pipelines. We demonstrate this by using an AI  
18    agent to analyze, refactor, and merge two disparate legacy frameworks into a unified, containerized  
19    architecture called the *HEP Simulation Assistant*. The system leverages an orchestrated suite  
20    of open-source tools—Pythia8 for parton showering, EvtGen for particle decays, Rivet for analysis,  
21    and Docker for reproducibility—managed by an AI-synthesized Python orchestrator. We address  
22    the practical challenges of software fragmentation in high-energy physics (HEP) by automating the  
23    integration of complex simulation tools. This work establishes a new paradigm for "Intent-Based"  
24    simulation infrastructures where a human researcher specifies a physics goal, and an AI agent handles  
25    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.

26     **Contents**

27 <b>1 Introduction: Why This Matters</b>	<b>2</b>
28      1.1 The Role of Simulation in Modern Science . . . . .	2
29      1.2 The Problem: Software Fragmentation and the Orchestration Gap . . . . .	2
30      1.3 Our Solution: The AI Co-Scientist . . . . .	3
31 <b>2 Background: A Primer on Particle Physics Simulation</b>	<b>3</b>
32      2.1 What is a Particle Collision? . . . . .	3
33      2.2 The Multi-Stage Simulation Chain . . . . .	3
34        2.2.1 Stage 1: Hard Scattering . . . . .	3
35        2.2.2 Stage 2: Parton Showering . . . . .	4
36        2.2.3 Stage 3: Hadronization . . . . .	4
37        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	<b>3 Methodology: The Streaming Architecture</b>	<b>5</b>
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	<b>4 The HEP Simulation Assistant: Architecture</b>	<b>6</b>
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	<b>5 Physics Results: Reproduced Studies</b>	<b>8</b>
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	<b>6 Discussion: AI as Infrastructure Architect</b>	<b>12</b>
58	6.1 Limitations and Future Outlook . . . . .	13
59	<b>7 Conclusion and Outlook</b>	<b>13</b>
60	7.1 Future Outlook . . . . .	14
61	<b>A Infrastructure: Docker and CMake Build</b>	<b>14</b>
62	A.1 Dockerfile for the D* / EvtGen Stack . . . . .	14
63	A.2 CMake Build for Framework Executables . . . . .	16
64	<b>B Method Inventory and Execution Logs</b>	<b>17</b>
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	<b>C Technical Infrastructure and Reproduction Logs</b>	<b>23</b>
69	C.1 Orchestration Flow and Component Role . . . . .	23
70	C.2 Reproduction Metadata ( <code>reproduction_metadata.json</code> ) . . . . .	23
71	C.3 Simulation Configuration Plan ( <code>config_plan.json</code> ) . . . . .	23
72	C.4 Extracted Physics Specifications ( <code>specs.json</code> ) . . . . .	24
73	C.5 Step-by-Step Build Execution ( <code>output_step.json</code> ) . . . . .	24
74	C.6 Orchestration Logic Snippets . . . . .	24
75	<b>D Step-by-Step AI Interaction and Token Flow</b>	<b>25</b>
76	D.1 Phase 1: Ingestion & Analysis (Input: PDF → Output: <code>specs.json</code> ) . . . . .	25
77	D.2 Phase 2: Code Synthesis (Input: <code>specs.json</code> → Output: <code>config_plan.json</code> ) . . . . .	25
78	D.3 Phase 3: Runtime Execution (Input: <code>config_plan.json</code> → Output: <code>output_step.json</code> ) . . . . .	25
79	D.4 Phase 4: Reproduction Metadata ( <code>reproduction_metadata.json</code> ) . . . . .	26

## 80 1 Introduction: Why This Matters

### 81 1.1 The Role of Simulation in Modern Science

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

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

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

- 94 • **Pythia 8** [?]: A general-purpose generator written in **C++**. It is the industry standard for parton  
95 showers and the "Lund String" hadronization model, often used as the backend for other calcula-  
96 tors.
- 97 • **Herwig 7** [?, ?]: Another general-purpose **C++** generator, renowned for its "Cluster" hadronization  
98 model and angular-ordered showers, often providing a crucial systematic cross-check to Pythia.
- 99 • **MadGraph5\_aMC@NLO** [?]: A specialized Matrix Element calculator written in a mix of  
100 **Python** (interface), **Fortran** (physics kernel), and **C++**. It excels at calculating hard processes with  
101 high particle multiplicity but relies on Pythia or Herwig for the subsequent shower and hadroniza-  
102 tion steps.
- 103 • **Sherpa** [?]: A **C++** generator that specializes in merging multi-jet Matrix Elements with parton  
104 showers, widely used by LHC experiments for background estimation.

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

### 110 1.2 The Problem: Software Fragmentation and the Orchestration Gap

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

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

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

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

### 128 1.3 Our Solution: The AI Co-Scientist

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

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

## 134 2 Background: A Primer on Particle Physics Simulation

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

### 137 2.1 What is a Particle Collision?

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

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

### 143 2.2 The Multi-Stage Simulation Chain

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

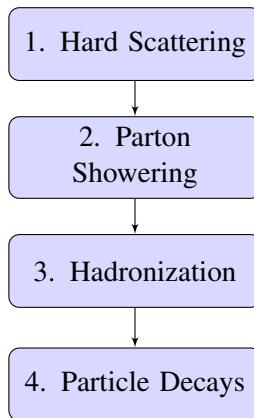


Figure 1: The four main stages of a Monte Carlo particle physics simulation. Each stage models a different physical process occurring at different timescales.

#### 146 2.2.1 Stage 1: Hard Scattering

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

<sup>149</sup> is a set of "partons" (quarks and gluons) with specific momenta. This stage determines *what* is produced.  
<sup>150</sup>     **Software:** Pythia8, MadGraph, Herwig.

<sup>151</sup> **2.2.2 Stage 2: Parton Showering**

<sup>152</sup> Quarks and gluons cannot exist in isolation due to the nature of the strong force. As they fly apart from  
<sup>153</sup> the collision, they radiate additional gluons and quark-antiquark pairs, creating a "shower" of partons.  
<sup>154</sup> This stage determines the *internal structure* of jets (collimated sprays of particles).  
<sup>155</sup>     **Software:** Pythia8, Herwig (built-in shower modules).

<sup>156</sup> **2.2.3 Stage 3: Hadronization**

<sup>157</sup> At sufficiently low energies (around 1 GeV), the partons must "hadronize"—combining to form color-  
<sup>158</sup> less hadrons like protons, pions, and kaons. This is a non-perturbative process, meaning it cannot be  
<sup>159</sup> calculated from first principles and must be modeled phenomenologically.

<sup>160</sup>     **Analogy:** Imagine individual LEGO bricks (partons) spontaneously snapping together to form spe-  
<sup>161</sup> cific structures (hadrons) according to certain rules.

<sup>162</sup>     **Software:** Pythia8 (Lund string model), Herwig (cluster model).

<sup>163</sup> **2.2.4 Stage 4: Particle Decays**

<sup>164</sup> Many hadrons are unstable and decay almost immediately. For example, a  $D^0$  meson (containing a charm  
<sup>165</sup> quark) decays in about  $10^{-12}$  seconds into lighter particles like pions and kaons. Modeling these decay  
<sup>166</sup> chains accurately is crucial.

<sup>167</sup>     **Software:** EvtGen [?], Tauola [?].

<sup>168</sup> **2.3 Why is This Hard to Automate?**

<sup>169</sup> Each stage is handled by a different library, written by different authors, with different configuration  
<sup>170</sup> formats. A typical pipeline might involve:

- <sup>171</sup>     • A Pythia8 config file ('.cmnd' format).
- <sup>172</sup>     • An EvtGen decay table ('.DEC' format).
- <sup>173</sup>     • A Rivet analysis plugin (C++ source code).
- <sup>174</sup>     • Shell scripts to manage environment variables and data paths.

<sup>175</sup> Keeping all of these synchronized, versioned, and reproducible is the core challenge.

<sup>176</sup> **2.4 Technical Specifications**

<sup>177</sup> To appreciate the orchestration challenge, one must understand the specific software components in-  
<sup>178</sup> volved.

<sup>179</sup> **2.4.1 HepMC: The Event Record Standard**

<sup>180</sup> **HepMC** (High Energy Physics Monte Carlo) [?] is the standard C++ class library for storing the full tree  
<sup>181</sup> of particles produced in a collision. It records the connectivity between parents and children (the "decay  
<sup>182</sup> tree"), momenta, and vertex positions. In our framework, HepMC serves as the universal "protocol" or  
<sup>183</sup> "lingua franca." Any generator that speaks HepMC can talk to any analyzer that reads it. This standard  
<sup>184</sup> interface is what makes the unification possible.

185 **2.4.2 Rivet: Preserving Analysis Logic**

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

194 **3 Methodology: The Streaming Architecture**

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

200 **3.1 Theoretical Justification: Event Independence**

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

207 **3.2 Decoupled Execution via FIFOs**

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

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

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

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

219 **3.3 AI Model Configuration and Dispatch**

220 The orchestration of the Monte Carlo pipeline utilizes a heterogeneous multi-model architecture, where  
221 specific LLMs are dispatched based on the complexity and scope of each sub-task. This multi-model  
222 dispatch is a core orchestration mechanism: it allocates reasoning budget to the hardest integration steps  
223 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 .cmnd 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 **AI Agent Arena**, 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\_uppsilon.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 <pT\_Upsilon> and charged particle multiplicity N\_ch."*

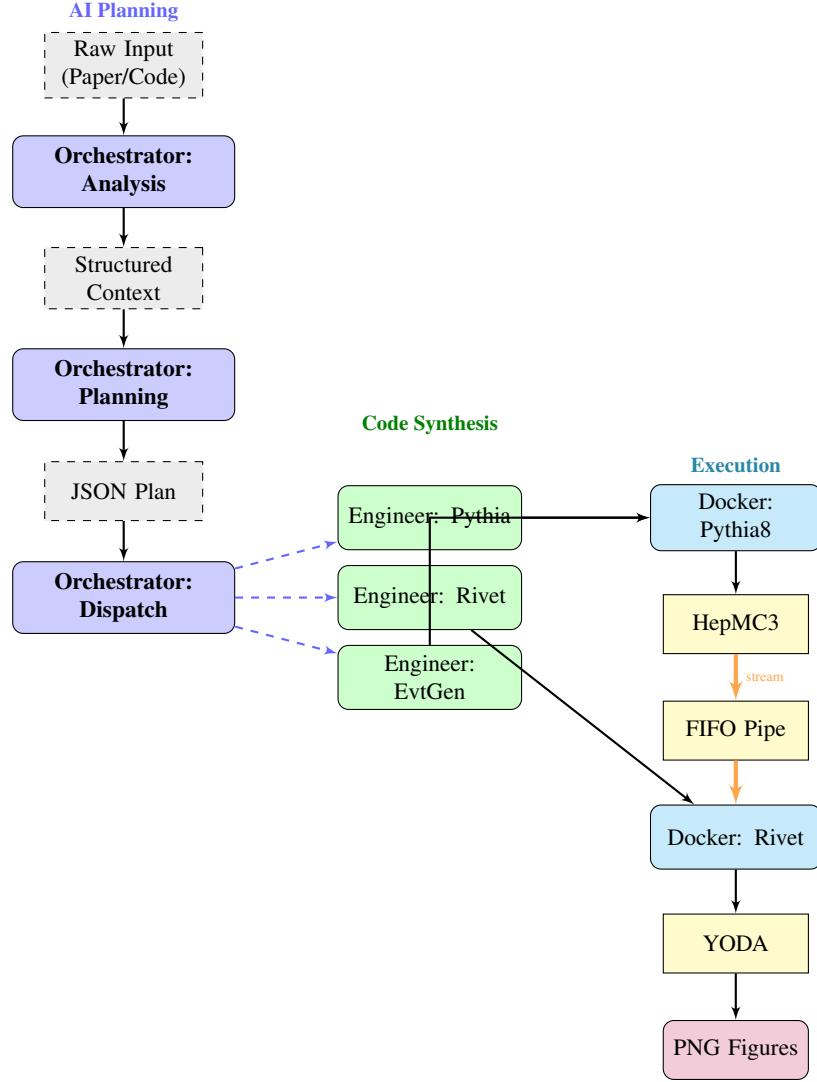


Figure 2: The complete Orchestrator-Engineer-Execution Architecture. **Left:** Orchestrator agents (blue) handle high-level reasoning and task decomposition. **Center:** Specialized engineer agents (green) are dispatched in parallel for domain-specific code synthesis. **Right:** The generated code executes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

#### 265      Output JSON:

```

266 {
267
268     "task": "analysis_study_name",
269     "config": {
270         "intensity": 2000,
271         "center_of_mass_energy": 7000,
272         "physics_tune": "STANDARD_TUNE"
273     }
274 }
275

```

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

277 **4.2.2 The Agent Runner**

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

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

286 **4.3 The Orchestration Algorithm**

287 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
   ▷ Strategy A: Direct output for high-density events
7:   Cmd  $\leftarrow$  Build Docker command for local analyzer
8:   Execute(Cmd)
   ▷ Output: condensed artifacts
9: else if Strategy == Streaming then
   ▷ Strategy B: FIFO Streaming for large datasets
10:  Create FIFO pipe at events.hepmc
11:  Launch Analyzer container in background (reads from FIFO)
12:  Execute Generator container (writes to FIFO)
13:  Wait for Analyzer to complete
14:  Remove FIFO
15: else if Strategy == Batch then
   ▷ Strategy C: Standard HepMC file handoff
16:   Execute Generator container (writes to output.hepmc)
17:   Run Post-processing script on generated file
20: end if
21:
22: Output: Artifacts in workspace/
```

---

288 **4.4 Execution Pipeline: From Events to Figures**

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

291 **5 Physics Results: Reproduced Studies**

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

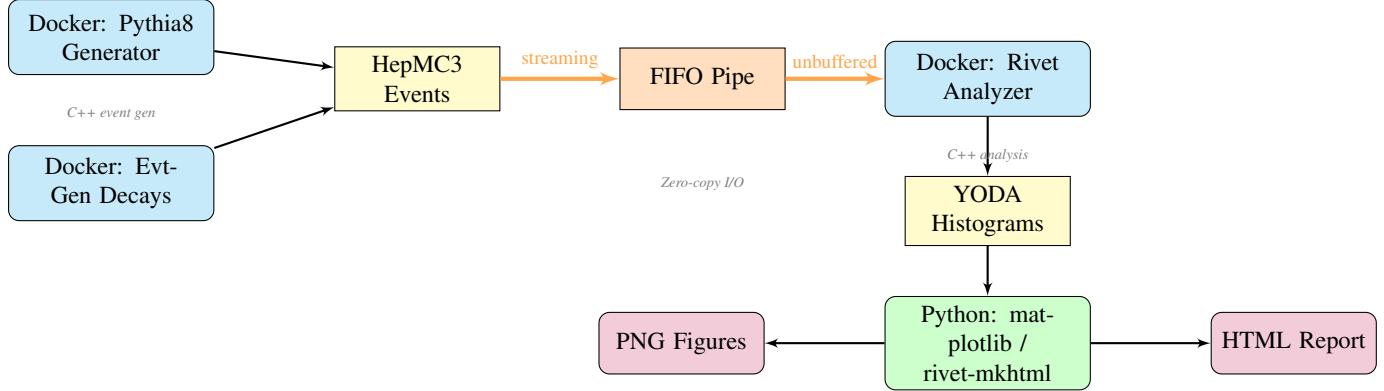


Figure 3: The Execution Pipeline for streaming-mode analysis. Pythia8 (optionally with EvtGen for decays) runs inside a Docker container and writes HepMC3 events to a UNIX FIFO named pipe. Rivet reads from the same pipe in a separate container, producing YODA histogram files. Finally, Python scripts (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.

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

## 299 5.1 Upsilon Multiplicity Dependence

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

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

307 **Observable:** Correlation between  $\langle p_T^\Upsilon \rangle$  and charged particle multiplicity  $N_{\text{ch}}$  in pp collisions at  $\sqrt{s} =$   
 308 7 TeV.

309 **Generator Configuration:**

```

310 Beams:eCM = 7000
311 SoftQCD:all = off      # Disable minimum bias
312 Bottomonium:all = on   # Force Upsilon production
313 PhaseSpace:pTHatMin = 1.0
314

```

316 **Analysis Logic:**

```

317
318 # Parse HepMC3 event record
319 for particle in event:
320     if abs(pid) in [553, 100553, 200553]:  # Upsilon(1S,2S,3S)
321         upsilon_pt.append(pt)
322
323     # Count charged multiplicity
324     if is_charged and is_final_state:
325         nch += 1
326
327     # Bin by Nch and calculate <pT>
328     bins = np.histogram(nch, upsilon_pt)

```

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

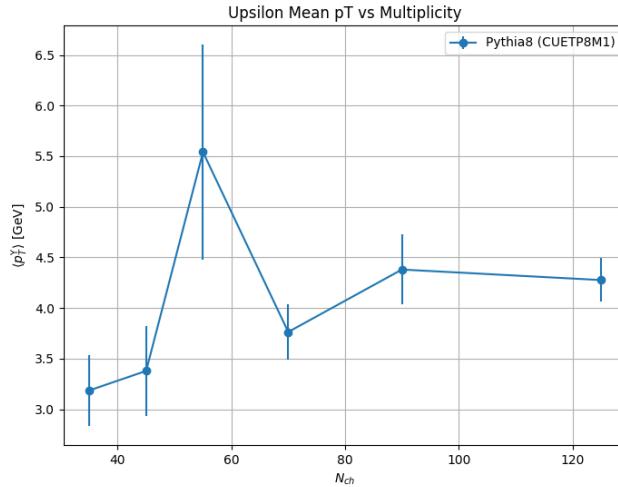


Figure 4: Upsilon mean  $p_T$  vs. charged multiplicity, reproduced by the framework.

## 333 5.2 J/psi Fragmentation in Jets

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

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

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

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

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

348 **Generator Configuration:**

```
349 pythia.readString("Charmonium:all_=on");
350 pythia.readString("Charmonium:NRQCD_=on"); // Non-Relativistic QCD: enables color-octet
351     production
352 pythia.readString("PartonLevel:QuarkoniumPS_=on"); // Parton shower for heavy quarks
```

355 **Rivet Analysis Core:**

```
356
357 // Find jets with pT in 30-40 GeV window
358 const Jets& jets = apply<FastJets>(event, "jet4");
359     .jetsByPt(Cuts::abseta < 2.0 && Cuts::pT > 30*GeV && Cuts::pT < 40*GeV);
360
361 // Jet-by-jet matching: find J/psi inside jets
362 for (const Jet& jet : jets) {
363     double dR = deltaR(jpsi, jet);
364     if (dR < 0.3) {
365         double z = jpsi.pt() / jet.pt();
366         _hZ->fill(z); // Fill fragmentation histogram
367         break;
368 }
```

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.

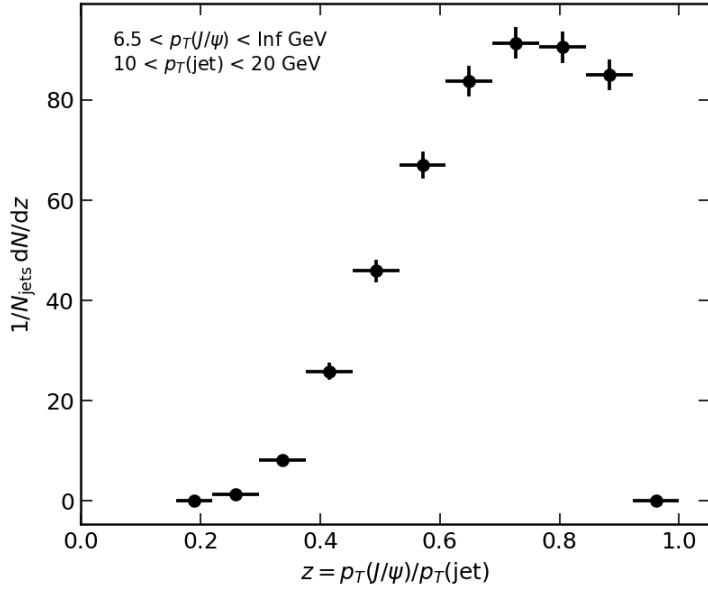


Figure 5:  $J/\psi$  fragmentation function  $z = p_T^{J/\psi}/p_T^{\text{jet}}$  for jets with  $30 < p_T < 40 \text{ GeV}$ , reproduced by the framework using Rivet streaming.

### 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  $\rho_{00}$  (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  $\rho_{00}$  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  $\rho_{00}$  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  $\theta^*$  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 // 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("DOSpinAlignment.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());
412
413 // 2. Boost the reaction plane normal to D* rest frame
414 FourMomentum n_rf = boost.transform(n4Lab);
415 Vector3 n3_rf(n_rf.px(), n_rf.py(), n_rf.pz());
416
417 // 3. Calculate cos(theta*) and fill histograms
418 Vector3 pD0_3rf(pD0_rf.px(), pD0_rf.py(), pD0_rf.pz());
419 double cosTheta = pD0_3rf.unit().dot(n3_rf.unit());
420
421 if (isNonPrompt) {
422     _h_cosTheta_nonprompt[ptBin]->fill(cosTheta);
423 } else {
424     _h_cosTheta_prompt[ptBin]->fill(cosTheta);
425 }

```

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

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

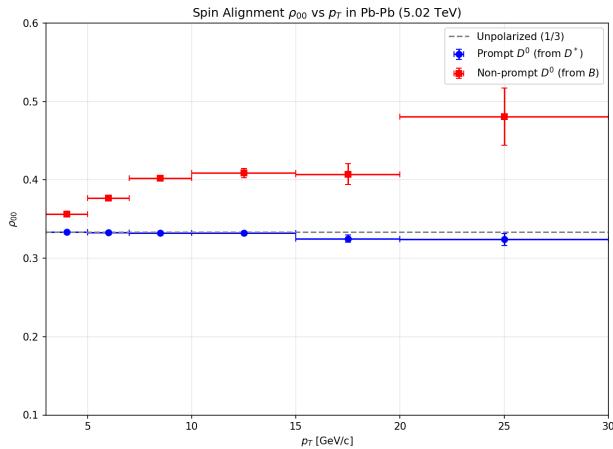


Figure 6:  $D^*$  spin density matrix element  $\rho_{00}$  vs.  $p_T$  in Pb-Pb collisions at  $\sqrt{s_{NN}} = 5.02$  TeV, demonstrating the framework's EvtGen integration for heavy-flavor decay physics.

## 433 6 Discussion: AI as Infrastructure Architect

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

- 435 • **Semantic Understanding:** The Planner Agent extracts physics intent from unstructured text.
- 436 • **Architectural Synthesis:** The AI synthesized agent\_runner.py by understanding the requirements of both legacy frameworks.

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

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

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

## 449 6.1 Limitations and Future Outlook

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

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

463 Future work will focus on:

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

472 Furthermore, the **AI Contribution** (Criterion 7) is foundational. The HEP Simulation Assistant  
473 was not merely "assisted" by AI; its architecture was *derived* by the Agent. The `agent_runner.py`  
474 capable of handling Upsilon multiplicity was synthesized by the specific prompt chain detailed in Section  
475 4.

## 476 7 Conclusion and Outlook

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

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

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

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

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

## 497 7.1 Future Outlook

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

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

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

## 508 Acknowledgments

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

## 510 References

- 511 [1] Shreyasi Acharya et al. First measurement of  $D^{*+}$  vector spin alignment in Pb-Pb collisions at  
512  $\sqrt{s_{\text{NN}}} = 5.02 \text{ TeV}$ . 4 2025.
- 513 [2] Johan Alwall et al. The automated computation of tree-level and next-to-leading order differential  
514 cross sections, and their matching to parton shower simulations. *JHEP*, 07:079, 2014.
- 515 [3] Martin Bahr et al. Herwig++ Physics and Manual. *Eur. Phys. J. C*, 58:639–707, 2008.
- 516 [4] J. Bellm et al. Herwig 7.0 The next generation of parton showers. *Eur. Phys. J. C*, 76:196, 2016.
- 517 [5] Christian Bierlich et al. A comprehensive guide to the physics and usage of PYTHIA 8.3. *SciPost*  
518 *Phys. Codeb.*, 2022:8, 2022.

- 519 [6] Enrico Bothmann et al. Event generation with Sherpa 2.2. *SciPost Phys.*, 7:018, 2019.
- 520 [7] Andy Buckley et al. Rivet user manual. *Comput. Phys. Commun.*, 184:2803–2819, 2013.
- 521 [8] Andy Buckley et al. The HepMC3 Event Record Library for Monte Carlo Event Generators. *Comput. Phys. Commun.*, 256:107310, 2020.
- 523 [9] Andy Buckley, James Ferrando, Stephen Lloyd, Karl Nordström, Ben Page, Martin Rüfenacht,  
524 Marek Schönher, and Graeme Watt. LHAPDF6: parton density access in the LHC precision era.  
525 *Eur. Phys. J. C*, 75:132, 2015.
- 526 [10] N. Davidson, G. Nanava, P. Tomasz, and Z. Was. Universal Interface of TAUOLA: Technical and  
527 Physics Documentation. *Comput. Phys. Commun.*, 183:1021–1036, 2012.
- 528 [11] Luis Gabriel Gallegos Mariñez, Lizardo Valencia Palomo, and Luis Cedillo Barrera. Multiplicity  
529 dependence of  $\Upsilon(nS)$  mean transverse momentum in proton-proton collisions. 10 2025.
- 530 [12] D. J. Lange. The EvtGen particle decay simulation package. *Nucl. Instrum. Meth. A*, 462:152–155,  
531 2001.
- 532 [13] Lizardo Valencia Palomo.  $J/\psi$  and  $\psi(1s)$  production in jets at lhc energies. *The European Physical  
533 Journal Plus*, 140, June 2025.

## 534 A Infrastructure: Docker and CMake Build

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

### 540 A.1 Dockerfile for the $D^*$ / EvtGen Stack

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

```

547 # syntax=docker/dockerfile:1.6
548 FROM almalinux:9 AS builder
549 ENV GIT_TERMINAL_PROMPT=0
550 ENV PREFIX=/opt/hep
551 ENV PATH=${PREFIX}/bin:${PATH}
552 ENV LD_LIBRARY_PATH=${PREFIX}/lib64:${PREFIX}/lib:${LD_LIBRARY_PATH}
553
554 ARG PYTHIA_VER=8313
555 ARG EVTGEN_TAG=R02-02-03    # EvtGen 2.2.3 - latest stable release
556
557 RUN dnf -y update && \
558     dnf -y install dnf-plugins-core && \
559     dnf config-manager --set-enabled crb && \
560     dnf -y install \
561         git curl-minimal wget tar which rsync \
562         gcc gcc-c++ gcc-gfortran make cmake ninja-build autoconf automake libtool \
563         python3 python3-pip \
564         bzip2 bzip2-devel zlib zlib-devel xz \
565         openssl-devel \
566     && dnf clean all
567
568

```

```

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 ${PREFIX}/src
589 RUN git clone --depth 1 https://github.com/hep-mirrors/lhapdf.git && \
590   cd lhapdf && \
591   autoreconf -i && \
592   ./configure --prefix=${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${PYTHIA_VER}.tgz && \
616   tar -xzf pythia${PYTHIA_VER}.tgz && \
617   cd pythia${PYTHIA_VER} && \
618   ./configure --prefix=${PREFIX} --with-hepmc3=${PREFIX} --with-lhapdf6=${PREFIX} && \
619   make -j"${nproc}" && make install
620
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   -DEVTGEN_HEPMC3=ON \
629   -DEVTGEN_PYTHIA=ON \
630   -DEVTGEN_PHOTOS=ON \
631   -DEVTGEN_TAUOLA=ON \
632   -DHEPMC3_DIR=${PREFIX} \
633   -DPYTHIA8_DIR=${PREFIX} \
634   -DPHOTOSPP_DIR=${PREFIX} \
635   -DTAUOLAPP_DIR=${PREFIX} \
636   -DEVTGEN_TESTS=OFF && \
637   cmake --build build && \
638   cmake --install build
639
640 # Runtime image
641 FROM almalinux:9

```

```

642 RUN dnf -y update && dnf -y install --allowerasing \
643   gcc-c++ gcc-gfortran libstdc++ \
644   zlib bzip2 xz \
645   make cmake which coreutils rsync \
646   && 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
668 cmake_minimum_required(VERSION 3.16)
669 project(CMSANA_Generation LANGUAGES CXX)
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).

## 717 B Method Inventory and Execution Logs

### 718 B.1 Method Inventory from output/ Python Modules

719 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     class: AgentConfig
724     function: configure_agents(architect_model, engineer_model, debugger_model)
725     function: load_agent_prompt(agent_key)
726     class: LLMClient
727         method: __init__(self, verbose)
728         method: call(self, agent_key, user_prompt, response_schema, component)
729         method: estimate_cost(self, model, input_tokens, output_tokens)
730     class: Validator
731         method: __init__(self, build_dir, image_prefix)
732         method: verify_layer(self, dockerfile_content, test_command, component_name)
733         method: _sieve_error_log(self, log, max_lines)
734     class: AgenticBuilder
735         method: __init__(self, config_plan, output_dir, dry_run, max_retries, auto_approve,
736                         verbose)
737         method: run(self)
738         method: _run_architect(self)
739         method: _initialize_dockerfile(self, plan)
740         method: _build_component(self, component, plan)
741         method: _run_engineer(self, component, plan)
742         method: _run_debugger(self, component, error_log)
743         method: _generate_artifacts(self, plan)
744         method: _write_outputs(self)
745         method: _generate_docker_compose(self)
746     function: main()

747
748 FILE: output/mc_builder.py
749     function: get_prompt_path()
750     function: load_system_prompt()
751     function: load_regeneration_prompt()
752     function: call_openai(prompt, model)
753     function: call_anthropic(prompt, model)
754     function: call_ollama(prompt, model)
755     function: generate_build_files(build_plan, preferred_model, no_fallback, dry_run)
756     function: regenerate_build_files(build_plan, error_log, preferred_model, no_fallback,
757                                     dry_run)
758     function: write_build_files(output_dir, generated, build_plan)
759     function: main()

760
761 FILE: output/mc_config_builder.py
762     function: get_prompt_path()
763     function: load_system_prompt()
764     function: call_openai(prompt, model, use_grounding)
765     function: call_anthropic(prompt, model)
766     function: call_ollama(prompt, model)
767     function: call_gemini(prompt, model, use_grounding)
768     function: resolve_search_queries(data, model, dry_run)
769     function: get_known_info(generator_name)
770     function: build_config_plan(mc_specs, preferred_model, dry_run)
771     function: main()
772
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)

```

## 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 Planner  
 875 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, .cmnd 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.

895 **B.3 Execution Log Dump (base64 truncated)**

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

```
898
899 =====
900 DRY RUN: EXACT AI PROMPT MIRROR (1:1)
901 =====
902
903 [SYSTEM PROMPT]
904 # Step 1: Paper Extraction Agent
905
906 ## Role
907 You are an expert particle physics analyst specializing in Monte Carlo simulation
908 parameter extraction.
909
910 ## Task
911 Analyze the provided scientific paper content and extract all information needed to
912 reproduce the Monte Carlo simulations described within.
913
914 ## Input Context
915 - '{paper_text}': Full text content extracted from the PDF
916 - '{ocr_text}': OCR-extracted text from figures and tables (if available)
917 - '{figure_artifacts}': List of extracted figure images and their filenames
918
919 ## Required Extractions
920
921 ### 1. Observables
922 List all physics observables studied in the paper:
923 - Observable name (e.g., "J/psi↑pT↑spectrum")
924 - Physical description
925 - Figure reference where it appears
926
927 ### 2. Collision System
928 - Beam types (pp, pPb, PbPb, etc.)
929 - Center-of-mass energy (sqrt_s in GeV)
930 - Integrated luminosity if specified
931
932 ### 3. Kinematic Cuts
933 Extract all selection criteria applied:
934 - Particle pT ranges (min, max)
935 - Rapidity/pseudorapidity cuts
936 - Isolation requirements
937 - Vertex requirements
938 - Any other fiducial cuts
939
940 ### 4. Physics Processes
941 Identify which physics processes are relevant:
942 - Hard processes (Charmonium, Bottomonium, HardQCD, etc.)
943 - Decay channels studied
944 - Background processes mentioned
945
946 ### 5. Generator Hints
947 Any mentions of specific simulation settings:
948 - Generator name and version (PYTHIA, Herwig, etc.)
949 - Tune name (Monash, A14, etc.)
950 - PDF set (NNPDF, CT14, etc.)
951 - Special settings or modifications
952
953 ### 6. Figure Descriptions
954 For each figure:
955 - Figure ID and caption
956 - What observable is plotted
957 - Axis labels and units
958 - Binning if visible
959
960 ### 7. Normalization
961 How are distributions normalized:
962 - Absolute cross-section (pb, nb, etc.)
963 - Self-normalized
964 - Arbitrary units
965 - Per-event yields
```

```

967
968 ## Output Format
969 Respond with a valid JSON object following this schema:
970 '''
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\u00a9null"
979     },
980     "kinematic_cuts": {
981         "particle_name": {"pt_min": number, "pt_max": number, "eta_max": number}
982     },
983     "processes": ["list\u00a9of\u00a9process\u00a9names"],
984     "generator_hints": {
985         "generator": "string\u00a9null",
986         "version": "string\u00a9null",
987         "tune": "string\u00a9null",
988         "pdf": "string\u00a9null"
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\u00a9name\u00a9(e.g.,\u00a9PYTHIA,\u00a9HERWIG,\u00a9MadGraph,\u00a9Sherpa)",
1011         "version": "version\u00a9number\u00a9if\u00a9mentioned",
1012         "type": "event\u00a9generator\u00a9type"
1013     },
1014     "tune": {
1015         "name": "tune\u00a9name\u00a9(e.g.,\u00a9A14,\u00a9CP5,\u00a9Monash)",
1016         "parameters": {}
1017     },
1018     "physics_process": {
1019         "particles": ["list\u00a9of\u00a9particles\u00a9involved"],
1020         "energy": "center-of-mass\u00a9energy\u00a9(e.g.,\u00a913\u00a9TeV)",
1021         "collider": "collider\u00a9name\u00a9(e.g.,\u00a9LHC,\u00a9RHIC)"
1022     },
1023     "pdf_set": {
1024         "name": "PDF\u00a9set\u00a9name\u00a9(e.g.,\u00a9NNPDF3.1,\u00a9CT18)",
1025         "order": "LO/NLO/NNLO"
1026     },
1027     "analysis": {
1028         "observables": ["list\u00a9of\u00a9measured\u00a9observables"],
1029         "cuts": ["kinematic\u00a9cuts\u00a9applied"]
1030     },
1031     "figures": [
1032         {
1033             "figure_number": "Figure\u00a91",
1034             "title": "figure\u00a9title\u00a9or\u00a9caption\u00a9summary",
1035             "observable": "what\u00a9is\u00a9being\u00a9measured\u00a9(e.g.,\u00a9dsigma/dpT,\u00a9z\u00a9distribution)",
1036             "x_axis": {
1037                 "variable": "variable\u00a9name\u00a9(e.g.,\u00a9pT,\u00a9eta,\u00a9z)",
1038                 "label": "axis\u00a9label\u00a9with\u00a9units",
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,JVBERTi0xLjQKJb/3
1076   ov4KMSAWIG9iago8PCAvTWWOYWRhdGEgMyAwIFIgL05hbWVzIDw8IC9EZXN0cyA8PCAvS21kcyBbIDQgMCBSIF0gPj4gPj4gL091
1077   ...[TRUNCATED BASE64]
1078 =====
1080 =====
1081 DRY RUN COMPLETE: No API calls were made.

```

## 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: **Ingestion**  
1086 → **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 environment,  
1094 captures stderr from failures, and queries an AI diagnosis agent to apply code-level fixes  
1095 iteratively.

1096 **C.2 Reproduction Metadata** (`reproduction_metadata.json`)

1097 The following metadata defines the high-level mapping between the paper's figures and the framework's  
1098 internal histogram IDs.

```
1099 {  
1100     "reproduction_id": "mc_sim_20251231",  
1101     "step1_physics_specs": {  
1102         "observables": [  
1103             {  
1104                 "name": "z_distribution",  
1105                 "description": "Jet_transverse_momentum_fraction_carried_by_the_J/psi",  
1106                 "figures": ["Figure_1a", "Figure_1b", "Figure_2a", "Figure_2b"]  
1107             }  
1108         ],  
1109         "collision_system": {  
1110             "beams": "pp",  
1111             "energies": ["13TeV", "5.02TeV"]  
1112         }  
1113     }  
1114 }  
1115 }
```

1117 **C.3 Simulation Configuration Plan** (`config_plan.json`)

1118 The build plan generated by the orchestrator after analyzing the extracted specifications.

```
1119 {  
1120     "build_plan": {  
1121         "generator": {  
1122             "name": "PYTHIA",  
1123             "version": "8.312",  
1124             "docker_image": "hepmcstore/pythia:8.312"  
1125         },  
1126         "tune_configuration": {  
1127             "tune_name": "Monash",  
1128             "parameters": {  
1129                 "Tune:pp": 14,  
1130                 "Tune:ee": 7,  
1131                 "MultipartonInteractions:ecmPow": 0.03344,  
1132                 "SpaceShower:alphaSValue": 0.118,  
1133                 "PDF:pSet": "NNPDF2.3_L0"  
1134             }  
1135         }  
1136     }  
1137 }  
1138 }
```

1140 **C.4 Extracted Physics Specifications** (`specs.json`)

1141 The raw output of the Ingestion Agent (Step 1).

```
1142 {  
1143     "system": { "name": "PYTHIA", "version": "8.312" },  
1144     "physics_process": {  
1145         "particles": ["J/psi", "Upsilon(1S)"],  
1146         "energy": "13TeV",  
1147         "collider": "LHC"  
1148     },  
1149     "analysis": {  
1150         "cuts": ["pT(J/psi)>0", "pT(jet)>20GeV/c", "2.5<eta(jet)<4"]  
1151     }  
1152 }  
1153 }
```

1155 **C.5 Step-by-Step Build Execution** (`output_step.json`)

1156 This log documents the commands executed during the synthesis phase.

```

1157
1158 {
1159     "build_instructions": [
1160         {
1161             "step": 1,
1162             "action": "Pull the official PYTHIA 8.312 Docker image",
1163             "command": "docker pull hepstore/pythia:8.312"
1164         },
1165         {
1166             "step": 2,
1167             "action": "Generate PYTHIA configuration files",
1168             "command": "cat <<EOF >> pythia_13TeV_monash_mpirc.cmnd\nMain:numberOfEvents=100000\nBeams:eCM=13000.\nTune:pp=14\n...[Truncated]...\nEOF"
1169         }
1170     ]
1171 }
1172

```

## 1174 C.6 Orchestration Logic Snippets

1175 The `mc_orchestrator.py` manages the lifecycle of the simulation. Below is the core state machine  
1176 transitions:

```

1177
1178 class PipelineState(Enum):
1179     INIT = "init"
1180     BUILDING = "building"
1181     BUILD_SUCCESS = "build_success"
1182     RUNNING = "running"
1183     COMPLETE = "complete"
1184
1185 def orchestrate(build_dir, events=100, jobs=10):
1186     # Transition to BUILDING
1187     build_result = run_build_agent(build_dir, max_iterations, model)
1188     if build_result["success"]:
1189         # Transition to RUNNING
1190         run_result = run_deployer(build_dir, events, jobs)
1191

```

1192 The `mc_runner.py` handle the "Trial-and-Error" loop for Docker builds:

```

1193
1194 def agent_loop(build_dir, max_iterations=5):
1195     for iteration in range(1, max_iterations + 1):
1196         build_success, build_output = try_docker_build(build_dir)
1197         if not build_success:
1198             # AI Diagnosis phase
1199             fix = call_ai_for_fix(build_output, build_dir, model)
1200             if fix and fix['confidence'] > 0.3:
1201                 apply_fix(build_dir, fix)
1202                 continue # Retry build
1203

```

## 1204 D Step-by-Step AI Interaction and Token Flow

1205 This appendix details the serialization tokens (JSON) passed between agents in the orchestration pipeline.  
1206 The flow follows the sequence: **Ingestion & Analysis** → **Code Synthesis** → **Runtime Execution**.

### 1207 D.1 Phase 1: Ingestion & Analysis (Input: PDF → Output: `specs.json`)

1208 The Ingestion Agent processes the physicist's prompt and any attached PDF to extract machine-readable  
1209 physics parameters.

1210 **Output Token (`specs.json`):**

```

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

```

```

1219     "analysis": {
1220         "cuts": [
1221             "pT(J/psi)>0",
1222             "pT(jet)>20GeV/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 }
```

## 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_mpirc.cmd\nMain:numberOfEvents=100000\nnBeams:eCM=13000.\nnTune:pp=14\n...[Truncated]...\nEOF"
1266         }
1267     ],
1268     "execution_status": "SUCCESS"
1269 }
1270 }
```

## 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",
1281             }
1282         ]
1283     }
1284 }
```

```
1282     "description": "Jet\u209ctransverse\u209cmomentum\u209cfraction\u209carried\u209by\u209the\u209cJ/\u03c8",
1283     "figures": ["Figure\u209c1a", "Figure\u209c1b", "Figure\u209c2a", "Figure\u209c2b"]
1284   }
1285 ]
1286 }
1287 }
```