



NextGen
Next Generation Triggers

Ultra-Ultra-Fast Flow Matching for On-the-Fly Event Generation

Summer Student Project

Author:

Koen Waggeveld

Supervisors:

Sioni Summers

Jaime León Holgado

Date:

August 29, 2026

Abstract

The Level-1 Scouting system introduced with the CMS Phase-2 upgrade captures reconstructed trigger objects at the full 40 MHz LHC collision rate. Stress-testing this system requires realistic input events sustained at this rate and delivered along the same path as real trigger data. Neither physics-based simulation nor generative models on external hardware can supply this. This work develops an on-the-fly generator of PUPPI kinematics that runs inside the Level-1 Trigger firmware itself, combining BDT-based flow matching with conifer’s translation of BDTs into FPGA firmware. Generators are trained on minimum-bias data at two levels of granularity: whole jets, and the individual PUPPI particles constituting them. The jet-level velocity field and the numerical integrator were synthesised and built on the `xcvu13p-flga2577-2-e` chip of the Phase-2 Level-1 Trigger. The design occupies 20.4 % of the device, achieves an initiation interval of one clock cycle at 360 MHz, and agrees bitwise with its HLS simulation. The on-chip generator sustains at least 22.5×10^6 minimum-bias events per second, just over half the collision rate from a single instance, establishing that every stage of a flow-matching sampler can be expressed and run in trigger firmware. The sampler reproduces the bulk of each marginal and the broad shape of the pairwise densities. The limiting factor differs per channel: untuned fixed-point precision for η and ϕ , and the learnt field for p_T , whose tail cuts off at 163 GeV. The constituent-level model raises the channel count from 3 to 49 and extrapolates to roughly 5.3 million LUTs, some three times the capacity of the chip; reducing this occupancy, for which BUFF’s multi-output ensembles are the natural candidate, is the principal step remaining, though the multi-output model trained here does not yet reproduce the target. The implementation is released as `puppibuff`.

Contents

1	Introduction	1
2	Theory	2
2.1	PUPPI kinematics	2
2.2	Flow matching	2
2.3	FPGAs	4
3	Methods	5
3.1	Datasets and encoding	5
3.2	Model and training	5
3.3	Firmware translation	6
3.4	Evaluation	6
4	Results	7
4.1	Jet-level generator	7
4.2	Hardware implementation	9
4.3	Constituent-level generator	10
5	Discussion	11
5.1	Fidelity of the jet-level generator	11
5.2	Hardware feasibility and throughput	13
5.3	Constituent-level generation on FPGAs	14
6	Conclusion	14
	Acknowledgements	15
	Appendices	16
A	Circular encoding of ϕ	16
B	Numerical integrators	17
C	Hyperparameter scan	19
D	The high- p_T tail	21
E	Multi-output field models	22
F	Choice of prior distribution	24
	Bibliography	25

1 Introduction

The LHC collides protons at 40 MHz, delivering events at rates far higher than can be read out and stored. The CMS Level-1 Trigger (L1T) system filters by accepting only a small fraction of events within a few microseconds of latency; everything else is discarded, along with all information about it. The Phase-2 upgrade for the High-Luminosity LHC implements a second look at this data with the Level-1 Scouting system. This system captures reconstructed particles and intermediate L1T data at the full 40 MHz rate for further analysis [1, 2]. This input stream is of limited resolution but complete, allowing analysis of processes previously not visible because they are too soft to cross any threshold or too abundant to record in full.

Stress-testing such a system requires a stream of realistic input data at the full collision rate, sustained for long periods. Traditional physics-based simulation cannot fulfil this task: full detector simulation and reconstruction take of order 20 s per event [3], roughly nine orders of magnitude short of the collision rate. Moreover, any sample large enough to test the system for a meaningful duration would be too large to store. Instead, an on-the-fly generator is needed that emits trigger events continuously at full collision rate.

One option for such a generator is a machine-learning based generative model, which samples far faster than a first-principles simulation. A study of end-to-end simulators based on normalizing flows shows sampling speeds up to $\mathcal{O}(1)$ MHz per object on a single GPU with its significantly more accurate model running over an order of magnitude slower [3]. An event consists of approximately 20 of such objects, resulting in event sampling at $\mathcal{O}(0.05)$ MHz. Reaching the nearly three order-of-magnitude speed up to the target 40 MHz by scaling hardware is expensive.

Throughput is not the only constraint. The Level-1 Scouting system takes its input directly from the L1T, and a stress test should mirror this path. This places the generator on the trigger firmware rather than on external hardware.

Two recent developments together make these problems solvable. The first is BUFF [4], which implements a flow-matching based model [5] using gradient-boosted decision trees (BDTs) in place of neural networks. This leverages the strength of BDTs on tabular data to reach high fidelity at a significantly lower computational cost and sampling time compared to traditional neural-network based flow-matching models. The second development is conifer [6], a library that translates trained BDTs into FPGA firmware, and which is already used in the CMS Level-1 trigger. Taken together they imply that every component of a flow-matching sampler, the velocity field, the solver that integrates it, and the decoding back to physical units, can be expressed in trigger firmware, and therefore that the generator can be made to run end-to-end on the chip.

This work develops that combination into a working generator of PUPPI kinematics, the pileup-suppressed particle candidates that the Level-1 trigger reconstructs as inputs for the Level-1 Scouting system. Flow-based generative models using BDTs are trained using flow matching on minimum-bias data at two levels of granularity: whole jets, and the individual PUPPI particles constituting them. The resulting jet sampler is synthesised into a single FPGA design carrying the field predictions and the ODE solver that integrates over them, with the decoding back to physical space written as a matching block alongside them. The implementation is released as `puppibuff` [7].

The remainder of this report is organised as follows. Section 2 introduces PUPPI, the flow-matching framework and the discretisation adopted here, and the properties of the target hardware that constrain the model. Section 3 describes the datasets, encodings, training procedure and firmware translation in detail, together with the metrics used to score the generated distributions. Section 4 presents the output of both generators, the resource usage and throughput of the synthesised jet-level design, and the resource argument implying the constituent-level model cannot yet be built on the device. Section 5 interprets these results, separating the inaccuracies

due to the learnt model from those originating from the fixed-point implementation, and Section 6 summarises what has been achieved and what remains to be done before such a generator can be used to stress-test the Level-1 Scouting system.

2 Theory

2.1 PUPPI kinematics

At the target luminosity of the High-Luminosity LHC, each bunch crossing contains 140 simultaneous proton-proton collisions, with plans to increase it to an ultimate instantaneous luminosity of 200 collisions [8]. Only one is of interest to any given analysis, which is the leading vertex. The remaining particles constitute pileup, and have to be suppressed before jets or missing-energy quantities can be reconstructed reliably. Pileup Per Particle Identification (PUPPI) [9] does so per particle and continuously: rather than selecting particular reconstructed particles, it assigns each a weight expressing how consistent its local neighbourhood is with having originated at the leading vertex, and its four-momentum is scaled by that weight. Candidates embedded in pileup-like surroundings are scaled towards zero, while those in hard-scatter-like surroundings are left untouched. The output is thus a reweighted particle list of the crossing, where pileup is suppressed. It is these PUPPI candidates that are sent to the L1 Scouting system.

Two distinct objects derived from this stream are modelled in this work. The first is a jet, a collimated spray of hadrons [10], reconstructed from PUPPI candidates and described by the three-vector (p_T, η, ϕ) ; we refer to these as *jet-level events*, noting that for the purpose of training one sample is one jet rather than one collision. The second is the internal structure of such a jet: the variable-length list of PUPPI particles clustered into it by the anti- k_T algorithm [10] with radius parameter $R = 0.4$, each particle again described by the same (p_T, η, ϕ) kinematics. We refer to these as *constituent-level events*.

2.2 Flow matching

Generative modelling is the task of drawing new samples from a distribution p_{data} that is not available in closed form, but only through a finite set of examples drawn from it. For our purposes, that distribution is the distribution of the samples described in Section 2.1: a jet, or the constituents of one.

A *flow model* is a particular class of generative model that starts from a known distribution p_0 and transforms it by solving the initial value problem

$$\frac{d}{dt}\psi_t(\mathbf{x}_0) = \mathbf{v}_t(\psi_t(\mathbf{x}_0)), \quad \psi_0(\mathbf{x}_0) = \mathbf{x}_0, \quad (1)$$

with $\mathbf{v}_t: \mathbb{R}^d \rightarrow \mathbb{R}^d$ a time-dependent velocity field defined for $t \in [0, 1]$ and $\mathbf{x}_0 \sim p_0$. For this work, we take $p_0 = \mathcal{N}(\mathbf{0}, \mathbf{1}_d)$, the standard Gaussian in $d = n_{\text{channels}}$ dimensions.

Its solution ψ_t is called the *flow*: it exists and is unique whenever $\mathbf{v}_t$ is continuously differentiable with a bounded derivative [11, Theorem 3]. Equivalently it can be written as

$$\mathbf{x}_t := \psi_t(\mathbf{x}_0) = \mathbf{x}_0 + \int_0^t \mathbf{v}_\tau(\mathbf{x}_\tau) d\tau, \quad (2)$$

so that a single initial condition $\mathbf{x}_0$ gives a trajectory $\mathbf{x}_t$ defined by the field.

Applied to the whole of $\mathbb{R}^d$, the flow transports the prior p_0 along a family of distributions $(p_T)_{t \in [0, 1]}$: if $\mathbf{x}_0 \sim p_0$ then $\mathbf{x}_t \sim p_T$. This family thus forms a probability path, generated by

the vector field $\mathbf{v}$. When the field is such that the path ends on the data, $p_1 = p_{\text{data}}$, the flow becomes our generative model:

$$\psi_1(\mathbf{x}_0 \sim p_0) \sim p_{\text{data}}.$$

However, note that no such field $\mathbf{v}$ is known in advance: if there is one, we can just integrate it directly. A flow model therefore parametrises one as $\mathbf{v}_t^\theta$ with fitting parameters θ , and fits it such that the flow ψ_t^θ it induces through Eq. (1) carries the noise onto the data: $\psi_1^\theta(\mathbf{x}_0) \sim p_{\text{data}}$ for $\mathbf{x}_0 \sim p_0$. Sampling from the model then means to evaluate that flow. For any field of practical interest it admits no closed form, so the ODE is integrated numerically in a finite number of discrete steps from $t = 0$ to $t = 1$, as described in Algorithm 1.

Algorithm 1 Sampling from a flow model.

Require: learnt velocity field $\mathbf{v}^\theta$, step count n_{steps} , integrator step σ .

```

1: Sample  $\mathbf{x} \sim p_0$ 
2:  $h \leftarrow 1/(n_{\text{steps}} - 1)$ ,  $t \leftarrow 0$ 
3: for step  $\in (1, \dots, n_{\text{steps}} - 1)$  do
4:    $\mathbf{x} \leftarrow \sigma(\mathbf{v}^\theta, \mathbf{x}, t, h)$  ▷ One integration step
5:    $t \leftarrow t + h$ 
6: end for
7: return  $\mathbf{x}$ , approximately distributed as  $p_{\text{data}}$ 
```

It remains to fit θ to get a good approximation $\mathbf{v}^\theta \approx \mathbf{v}$. The typical machine learning approach of generating samples and applying gradient descent according to a loss function is not feasible in this context, as it would require backpropagating through the integrator and each of its field evaluations. *Flow matching* solves this by regressing onto the velocity field directly, which never requires generating a sample during training at all.

Suppose that a target field $\mathbf{v}_t$ generating a path from p_0 to p_{data} were available. Fitting the model to it would then be a regression problem with the flow matching objective

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{t \sim \mathcal{U}[0,1], \mathbf{x} \sim p_T} \left\| \mathbf{v}_t^\theta(\mathbf{x}) - \mathbf{v}_t(\mathbf{x}) \right\|^2. \quad (3)$$

As written, however, this is of little use: neither the target $\mathbf{v}_t$ nor the distribution p_T from which the states are drawn is known.

The way out is to condition on what we do have and using the law of total probability. A path from noise to the whole of p_{data} cannot be written down, but a path between one noise draw and one data sample can. We therefore condition on the pair $\mathbf{z} = (\mathbf{x}_0, \mathbf{x}_1) \sim p_0 \times p_{\text{data}}$ which we can construct, and recover the marginal quantities afterwards by integrating over $\mathbf{z}$.

Which conditional path to take is a modelling choice, and many are possible. We take the simplest option: the straight, deterministic line between the two endpoints of $\mathbf{z}$. The conditional distribution is then the point mass

$$p_T(\mathbf{x} \mid \mathbf{z}) = \delta(\mathbf{x} - [t\mathbf{x}_1 + (1-t)\mathbf{x}_0]) \quad (4)$$

such that any $\mathbf{x} \sim p_T(\cdot \mid \mathbf{z})$ equals $\mathbf{x}_t = t\mathbf{x}_1 + (1-t)\mathbf{x}_0$ with probability one. This gives the conditional velocity

$$\mathbf{v}_t(\mathbf{x} \mid \mathbf{z}) = \frac{d\mathbf{x}_t}{dt} = \mathbf{x}_1 - \mathbf{x}_0, \quad (5)$$

constant through time.

Using the law of total probability, we marginalise over $\mathbf{z}$ to obtain a path running from p_0 to p_{data} together with the field that generates it [5, 11]:

$$p_{\text{T}}(\mathbf{x}) = \mathbb{E}_{\mathbf{z} \sim p_0 \times p_{\text{data}}} [p_{\text{T}}(\mathbf{x} | \mathbf{z})], \quad (6)$$

$$\mathbf{v}_t(\mathbf{x}) = \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z} | \mathbf{x}_t = \mathbf{x})} [\mathbf{v}_t(\mathbf{x} | \mathbf{z})] \quad (7)$$

$$= \mathbb{E}[\mathbf{x}_1 - \mathbf{x}_0 | \mathbf{x}_t = \mathbf{x}]. \quad (8)$$

Replacing the marginal target of Eq. (3) with the conditional one of Eq. (5) gives the *conditional* flow matching objective

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t \sim \mathcal{U}[0,1], \mathbf{z} \sim p_0 \times p_{\text{data}}} \left\| \mathbf{v}_t^\theta(\mathbf{x}_t) - (\mathbf{x}_1 - \mathbf{x}_0) \right\|^2, \quad (9)$$

of which we have every term available. The two objectives differ only by a constant independent of θ , $\mathcal{L}_{\text{FM}}(\theta) = \mathcal{L}_{\text{CFM}}(\theta) + C$ [5, Theorem 2], so they have equal gradients and are minimised by the same field.

What remains is to choose the model. Conventionally, $\mathbf{v}^\theta$ is a neural network that takes the state $\mathbf{x}_t$ and time t as its inputs, trained by gradient descent on Eq. (9) with t drawn uniformly at each step. Following BUFF [4] we use a grid of BDTs instead, which requires t to be discretised onto n_{steps} time steps

$$t_s = \frac{s}{n_{\text{steps}} - 1}, \quad s = 0, 1, \dots, n_{\text{steps}} - 1, \quad (10)$$

with an independent regressor fitted at each, so that t is never an input feature. The expectation over t in Eq. (9) thereby becomes a sum over the grid, and since regressors at different t_k are independent, that sum separates into individual least-squares problems. The squared norm separates in the same way over the n_{channels} components of the velocity, leaving one scalar regression per time step s and channel c with objective

$$\mathcal{L}_{s,c}(\theta_{s,c}) = \mathbb{E}_{\mathbf{z} \sim p_0 \times p_{\text{data}}} \left[v_{s,c}^{\theta_{s,c}}(\mathbf{x}_{t_s}) - (\mathbf{x}_1 - \mathbf{x}_0)_c \right]^2. \quad (11)$$

Every BDT in the grid takes the full state $\mathbf{x}_{t_s} \in \mathbb{R}^{n_{\text{channels}}}$ as input and predicts a single component of the velocity there, so that the flow-matching model as a whole is a grid of $n_{\text{steps}} \times n_{\text{channels}}$ trained BDTs. We call the n_{channels} BDTs sharing a time step a *field step*: they jointly predict $\mathbf{v}_{t_s}$. To sample from p_{data} , we integrate over the field steps from a noise sample from the prior.

2.3 FPGAs

A field-programmable gate array (FPGA) is a chip made of reconfigurable logic primitives, namely lookup tables (LUTs), flip-flops (FFs), and digital signal processing blocks (DSPs). A design consumes a portion of each, and the resource that runs out first is what bounds it. The Level-1 Trigger is built on FPGAs, so a generator on the same type of chip outputting the same objects over the same interface as the Trigger is, from Level-1 Scouting’s perspective, indistinguishable from the Trigger’s interface.

Firmware is not written as a program but as a description of hardware, and converting software to hardware consists of two steps. First, High-Level Synthesis (HLS) compiles annotated C++ into a register-transfer level description. The formed blocks are then instantiated and wired together in VHDL, a hardware description language, to form the top-level design. Second, Vivado, the toolchain for the target FPGA, synthesises this VHDL into a netlist of the primitives above and implements it, placing and routing them onto the physical chip. Only after implementation

are the resource usage and the performance known: the estimates HLS reports beforehand are indicative and tend to overestimate.

The FPGA this work targets is the `xcvu13p-flga2577-2-e` chip, containing 1 728 000 LUTs, 3 456 000 FFs and 12 288 DSPs [12].

Two hardware design choices should be for the on-chip model design. Firstly the quantisation width, as arithmetic is fixed-point rather than floating-point. These widths are denoted as `ap_fixed<W, I>` types for a total of W bits of which I are integer. Increasing these for more capacity or resolution costs more resources, so the widths are a design parameter traded against numerical accuracy. Secondly, loops can be unrolled by wiring several identical blocks in parallel instead of passing each iteration through the same block. This trades resource usage for increase in throughput, which is the target performance for this work.

Resource usage scales with the total number of tree nodes instantiated, and hence with the $n_{\text{steps}} \times n_{\text{channels}}$ extent of the BDT grid of Section 2.2. The design objective is therefore not the most accurate model but the smallest one that still reproduces the target distribution acceptably, which is a constraint that recurs throughout what follows.

3 Methods

The generator described below follows BUFF [4] in predicting the time-discretised velocity field of Section 2.2 using a grid of BDTs, but deviates from its approach in the encoding of constituent multiplicity, and in the choice of numerical integrator. Both are explained below. The implementation is released as `puppibuff` [7].

3.1 Datasets and encoding

Both generators are trained on a sample of simulated minimum-bias data. The jet-level dataset holds 29 057 185 PUPPI jets, each described by a single kinematic (p_T, η, ϕ) triplet. The constituent-level dataset describes the same jets by their internal structure, as the PUPPI particles clustered by anti- k_T with $R = 0.4$ as in Section 2.1, each particle carrying the same three kinematic features.

The two representations share one encoding, in which features are normalised per channel before training, motivating the choice of the standard Gaussian prior of Section 2.2. Since the p_T channel has a high peak, p_T is `log1p`-transformed and then standardised, while η and ϕ are standardised directly. Although ϕ formally lives on a circle, we did not preserve this topology in the encoding: a circular $\phi \mapsto (\sin \phi, \cos \phi)$ encoding adds a channel and so increases model size, and it complicates the decoding. See Appendix A for a more elaborate discussion.

Constituent-level samples additionally need a fixed-shape tabular representation, obtained by truncating and zero-padding the p_T -sorted jagged arrays to fixed length $M = 16$, consistent with the L1T jet reconstruction. It remains to encode which particles are genuine. We considered two strategies: a per-constituent binary flag distinguishing real and padded entries, following BUFF, and a single per-jet scalar giving the true constituent count m . We adopted the latter as it scales better, adding one feature per jet rather than one per constituent particle. Lastly, the scalar multiplicity m is encoded like p_T : passed through `log1p` and standardised thereafter. A constituent-level sample is thus a vector of $16 \times 3 + 1 = 49$ channels against the 3 of a jet-level sample.

3.2 Model and training

With the target events encoded as $\mathbf{x}_1$, the training set is built by constructing paths from Gaussian noise $\mathbf{x}_0$ to the target data by linear interpolation as in Eq. (4), and discretising the flow-matching

time parameter into $n_{\text{steps}} = 15$ steps using Eq. (10).

At each such time step the field step is trained on the objective of Eq. (11): one XGBoost regressor per channel, each taking the interpolated state $\mathbf{x}_{t_s}$ as input and predicting its channel's component of the velocity there. Every regressor in the grid carries the same configuration: 50 trees of maximum depth 6, learning rate 0.1, L1 and L2 regularisation terms of 0.2 and 0.1, and the squared-error objective. These values and n_{steps} are shared by the jet-level and constituent-level model, but were fixed before the hyperparameter scan of Appendix C had finished. That gridsearch suggests a different point of the grid as the optimum.

For the integrator step σ of Algorithm 1 we chose the two-step Adams–Bashforth method (AB(2)) instead of the midpoint rule used in BUFF, for its lower per-step cost of one BDT inference per channel rather than two [13]. Moreover, as AB(2) is an explicit multistep method, previous predictions can be reused to increase sampling fidelity at equal step count, see Appendix B.

Once generated, samples are decoded by reversing the encoding. The p_T channels are scaled back to their original mean and variance and then passed through `expm1`, while η and ϕ are denormalised directly. The resulting p_T and η are clipped to the ranges observed in the training data, and ϕ is 2π -wrapped to $[-\pi, \pi)$. The scalar multiplicity m is decoded like p_T and, after rounding, clipped to $[0, M]$, so that the leading m particles of the jet are taken as real and the remaining slots discarded as padding. What leaves the decoder is a sample of the target distribution in physical space.

Only the jet-level model was carried through to firmware. The constituent-level model was trained and evaluated in Python alone; the resource argument for why it was not built is given in Section 4.3.

3.3 Firmware translation

Each field step of the jet-level model is translated to HLS as a separate block so that the steps can be synthesised in parallel. The AB(2) integration step and the decoding block including an `expm1` lookup table are written alongside them. Instances of these blocks are then tied together in a VHDL payload to form the numerical integrator, and hence the full sampler, as described in Section 2.3.

The individual BDTs constituting a field step are written with `conifer` version 1.9 [6, 14]. Since `conifer` emits each model as a stand-alone project whose sources all live at global scope, multiple BDTs cannot be combined into a single design. We therefore write the per-BDT firmware ourselves, adopting `conifer`'s tree implementation but placing each BDT's tree tables and the surrounding wrapper in a namespace of its own.

`Conifer`'s default `ap_fixed<18, 8>` precision is used for the BDT I/O. Between inferences a wider `ap_fixed<32, 8>` carries the integration steps, and `ap_fixed<24, 12>` the decoded samples. These quantisations are untuned given the limited time, and were left at these values.

3.4 Evaluation

The output distribution of a trained model is evaluated using two complementary metrics. The one-dimensional Wasserstein-1 (W_1) metric quantifies the distance between generated and target marginal distribution for each kinematic feature (p_T, η, ϕ). By construction this distance is blind to correlations between channels. To capture those as well we use the sliced Wasserstein (SW_1) distance, a cheap approximation of the classical multidimensional Wasserstein metric obtained by averaging W_1 over random projection directions [15]. We average over 128 projections of 500 000 events drawn from each distribution, with the channels standardised to the target's mean and variance before projecting so that the larger scale of p_T does not dominate the projection

directions. SW_1 is therefore reported in standardised units and is not on the same scale as the per-channel W_1 .

Both metrics are estimated from finite samples, so neither reach zero. To give a reference against which a score can be read, we compute the distance between the target and a disjoint 10^6 -event subsample of it. This value is the smallest distance the metric can return at that sample size, attributable entirely to the estimator rather than to the model, and we refer to it as the *floor*. A second reference is obtained by decoding 10^6 draws from the prior p_0 without integrating them, mimicking an untrained sampler. Together, the two bound the range within which a model's score is to be interpreted.

Visually, per-channel agreement is shown as histograms and inter-channel correlations as 2D kernel density estimate (KDE) contour plots for each channel pair, following the presentation of BUFF. Histograms are computed with 75 bins.

4 Results

In the following section, the output and performance of the jet-level and constituent-level PUPPI jet generators are presented. 10^6 samples were generated for each.

4.1 Jet-level generator

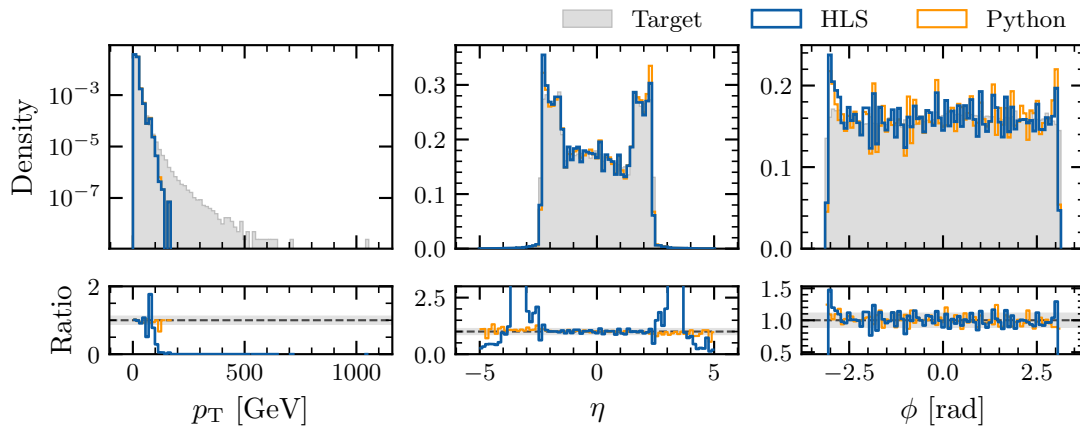


Figure 1: Marginal distributions of the 10^6 samples produced by HLS C++ and Python alongside the target distributions. Ratio column shows fractional agreement between HLS and target in blue, and HLS and Python in orange. Histogram built with 75 bins.

The marginal distributions of the generated jets are shown in Figure 1, alongside the target distribution and the corresponding samples drawn through the firmware implementation in HLS discussed in Section 4.2. The output p_T distribution follows the low- p_T peak well, but cuts off at 163 GeV while the training dataset has a maximum observed p_T of 1055.25 GeV. η is well modelled through the bulk, agreeing with the target to within a few percent out to $|\eta| = 2$, but overshoots at the peaks around $|\eta| \approx 1.5$. Lastly, the distributions for ϕ peak at the boundaries $\pm\pi$ but otherwise follow a noisy yet uniform distribution.

Figure 1 additionally carries the same model sampling in Python, using floating-point arithmetic. The two models are integrated from identical noise, with the only thing separating them the arithmetic the field evaluations and integration steps are carried out in. That is enough to separate them visually.

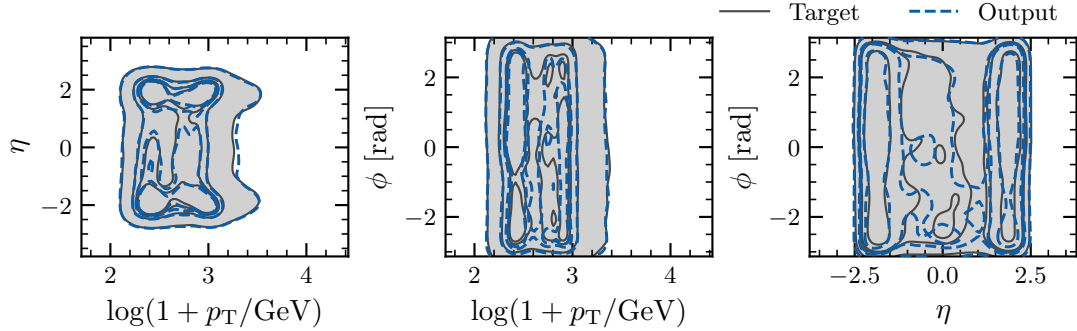


Figure 2: Pairwise 2D Gaussian KDE contours of the target distribution and output distribution of the HLS samples. Contours enclose 25 %, 50 %, 75 % and 95 % of observed probability mass.

Per-channel histograms are blind to correlations between channels, which is what the model ought to reproduce. Figure 2 shows the joint densities of each channel pair, displayed as 2D contours of the Gaussian KDE at 25 %, 50 %, 75 % and 95 % of the probability mass. The HLS output contours roughly follow the expected shape of the target contour, but fail to capture the finer detail of the landscape.

	$W_1(p_T)$	$W_1(\eta)$	$W_1(\phi)$	SW_1
HLS output	0.230	0.0428	0.0509	0.0242
Python output	0.147	0.00913	0.00675	0.00573
Decoded prior	0.832	0.308	0.242	0.0864
Held-out target	0.0147	0.00129	0.00140	0.00278

Table 1: Wasserstein distances to the target. Every row is measured against the same target of 28 057 185 jets, which is the target with the held-out probe removed. HLS output is 10^6 jets sampled by HLS C++. Python output is the same trained grid integrated from the same noise in floating-point representation, so the difference between the first two rows is the cost of the fixed-point implementation. Decoded prior and Held-out target are the two references of Section 3.4, each over 10^6 jets, and SW_1 uses the parameters given there as well.

The agreement between target and output distribution is quantitatively summarised in Table 1, along with the two references outlined in Section 3. The HLS output improves on the prior by a factor 3.6 to 7.2 across the three channels, and by 3.6 on SW_1 , but remains 16 and 36 times higher than the floor. Sampled using floating-point arithmetic the same model reaches 5 to 10 times the floor instead, and 2.1 on SW_1 against HLS’s 8.7.

The per-channel performances are limited by different factors. For η and ϕ , floating-point implementation improves the HLS implementation by a factor 4.7 on η , 7.5 on ϕ and 1.6 on p_T . The fixed-point arithmetic thus increases distance of these distributions from the target. For p_T , however, the picture is opposite: the floating-point model already sits 10 times above the floor, and quantisation adds a factor 1.6 on top. The p_T discrepancy is thus a property of the model, consistent with the tail cut-off of Figure 1, whereas the η and ϕ distances are a result of the quantisation.

The model reported here is the one that was translated to firmware. A hyperparameter scan performed afterwards indicated a different optimum, but the time required to synthesise and build the design meant that the implemented model had to be fixed before that scan could be completed. The two therefore do not coincide. Appendix C reports the scan, ranking its 32

configurations by SW_1 with a proxy for their on-chip size.

4.2 Hardware implementation

The design was implemented on the `xcvu13p-flga2577-2-e` FPGA, the chip used by the Phase-2 Level-1 Trigger, and synthesis was done using Vivado 2024.1. Of the $n_{\text{steps}} = 15$ trained field steps, 14 are instantiated as AB(2) evaluates the field at the left end of each step. The final step is never used as a result, so its BDTs do not need to be synthesised at all.

The field prediction and ODE solver were built on chip; specifically, 14 field steps and the 14 AB(2) updates that wire them into the integrator. The prior $\mathbf{x}_0$ is not generated on the device but drawn externally and given as input. At the output, the decoding block of Section 3 was written and synthesised but excluded from this build due to time constraints. Samples therefore leave the device in encoded space.

The correctness of the build was verified by comparing the output of the built design on the FPGA against the C++ HLS simulation of the same design by integrating 1024 identical noise samples, which agreed bitwise for all. The HLS results in Section 4.1 thus extend to the output of the FPGA.

Block	LUTs	FFs	DSPs	Device [%]
Mean field step	23 282	5571	0	1.3
Sum of 14 field steps	325 941	77 988	0	18.9
Mean AB(2) step	0	275	6	0.0
Sum of 14 AB(2) steps	0	3848	84	0.0
All blocks	327 072	85 490	84	18.9
Full design	351 931	122 624	84	20.4

Table 2: Resource usage of the built jet-level model with $n_{\text{steps}} = 15$, $d_{\text{max}} = 6$ and $n_{\text{trees}} = 50$ using AB(2) integration steps. Relative device utilisation on the `xcvu13p-flga2577-2-e` chip is reported for LUTs as they have the highest relative usage. Single-step rows are the mean over the 14 instantiated steps, which differ by less than 4 %.

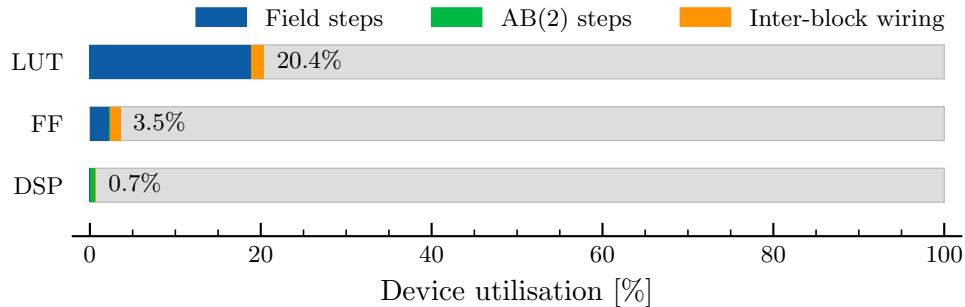


Figure 3: Relative resource usage of the jet-level generator on the `xcvu13p-flga2577-2-e` chip, corresponding to Table 2. LUTs are used most heavily.

The resource usage of the final build is given in Table 2 and shown against the capacity of the device in Figure 3. A single field step costs 23 282 LUTs on average, and the 14 instances together account for 99.7 % of the design’s 327 072 LUTs, or 18.9 % of the device. The AB(2) update is negligible as it requires no LUTs and 6 DSPs per instance. Including the surrounding inter-block wiring, the complete design occupies 351 931 LUTs (20.4 %), 122 624 FFs (3.5 %) and

84 DSPs (0.7%). The generator therefore fits comfortably within a single device, and LUTs are the binding resource throughout.

Figure 4 shows the implemented floorplan, in which the fourteen field steps are visible as distinct regions of nearly equal size.

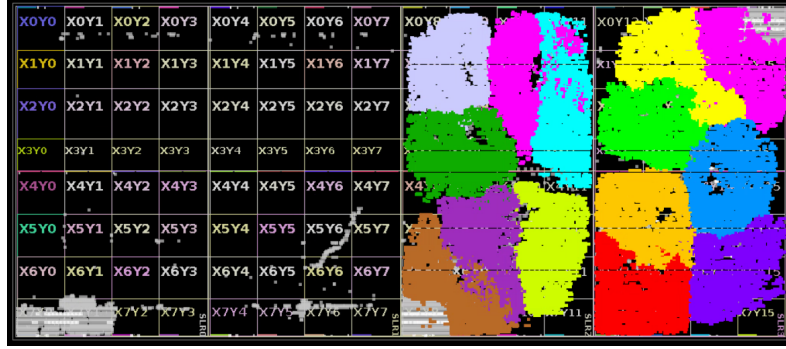


Figure 4: Floorplan of the built sampler on the xcvu13p-flga2577-2-e chip. Each of the 14 coloured regions represents one field step. The white regions in the top right and bottom left are I/O. Note that the chip looks approximately 50 % filled, in contradiction with Figure 3. This is an artifact resulting from the highlighting.

Every instantiated block is fully pipelined and achieved an initiation interval of one clock cycle at a clock period of 2.77 ns. The design thus generates one jet per cycle at 360 MHz, or 36×10^7 jets per second. With an upper bound of 16 jets per minimum-bias event, the generator can sustain at least 22.5×10^6 events per second. At the mean of 15.5 jets per minimum-bias event observed in the training data it corresponds to 23.2×10^6 . Either way, the design achieves just over half the 40 MHz collision rate. The full sampling has a latency of 128 cycles excluding decoding. The decoding block has a latency of 61 cycles predicted by Vitis HLS.

The resource estimates produced by the HLS strongly overestimate the usage for this design: a field step is estimated at 121 568 LUTs before building against the 23 282 it actually occupies, overshooting by a factor 5.2. Summing the synthesis estimates would suggest the design nearly fills the device, which is not the case.

The decoding block described in Section 3 was not included in the build due to time constraints. Despite this, the results above still hold, as the HLS estimates show its size to be small compared to the sampler: 3588 LUTs, 3845 FFs, and 17 DSPs. Using these HLS estimates, which are a pessimistic upper bound as argued before, it would result in an occupancy of 20.6 %, 3.7 % and 0.8 % of LUTs, FFs, and DSPs, respectively. The design would thus still easily fit on the chip.

4.3 Constituent-level generator

The constituent-level generator was trained and evaluated in Python only. The marginal output distributions of the pooled constituents are shown in Figure 5, together with the multiplicity distribution of the number of particles per jet. While the constituent-level model hyperparameters are equal to those of the jet-level model in Figure 1, it achieves lower fidelity. The p_T cutoff moves from 163 GeV to 66.2 GeV while η and ϕ obtain a peak at the origin.

The Wasserstein distances between target and output distributions are summarised in Table 3, alongside the floor and prior reference values explained in Section 3.4. The reference dataset here is larger than that of Table 1 as each jet now consists of up to 16 constituents. Consequently, the floor is lower and the ratios between the two tables should not be compared directly.

The reason this model was not built on hardware is its size. Representing a jet by its $M = 16$ constituents requires $16 \times 3 + 1 = 49$ channels instead of 3 of the jet-level model. As each field

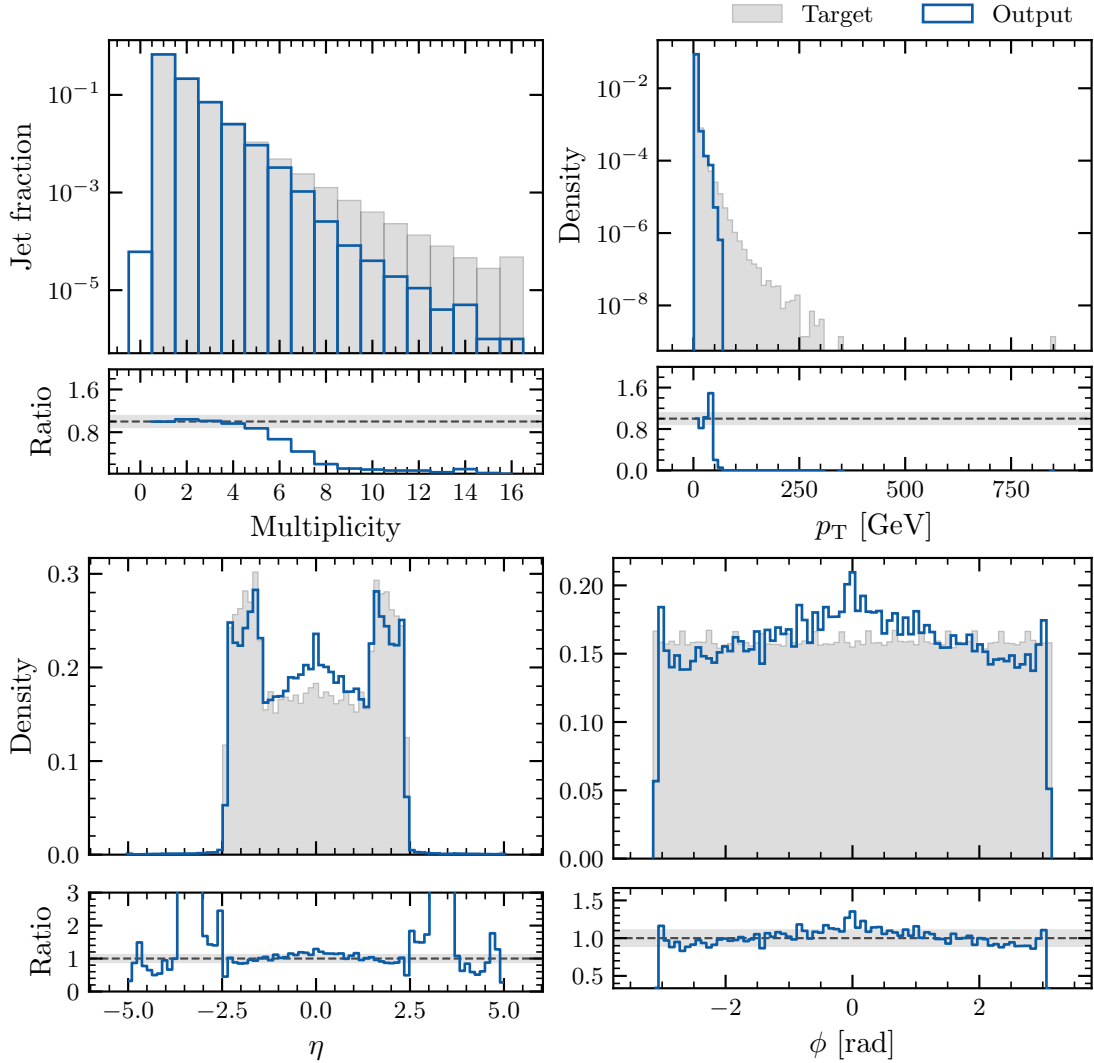


Figure 5: Multiplicity distribution and marginal distributions of the pooled constituents in the 10⁶ jet samples generated by Python, alongside the target distributions. Hyperparameters are the same as the model displayed in Figure 1.

step instantiates one BDT per channel, the increased channel count scales the implemented cost of a field step by 49/3 when accounting for number of BDTs alone. This results in roughly 5.3 million LUTs for the sampler, already roughly three times the capacity of the chip.

This is a lower bound for two reasons. Firstly, the per-tree resource cost itself grows with the number of channels, since each node of a tree selects its split feature from the full input vector. Moreover, the tree configuration used at jet level does not reach comparable accuracy on the constituent-level task, so a fair comparison would require deeper and more trees, increasing the cost further. A constituent-level generator of this form therefore does not fit on a single chip.

5 Discussion

5.1 Fidelity of the jet-level generator

The jet-level model build shows that a flow-matching generative model can be expressed in its entirety on an xcvu13p-f1ga2577-2-e FPGA. The velocity field, the solver that integrates it, and

	$W_1(m)$	$W_1(p_T)$	$W_1(\eta)$	$W_1(\phi)$	SW_1
Model output	0.0384	0.0975	0.0834	0.0928	0.0421
Decoded prior	0.262	0.553	0.299	0.247	0.113
Held-out target	0.00177	0.00426	0.00283	0.00113	0.00272

Table 3: Wasserstein distances to the target for the constituent-level generator. The rows and the SW_1 parameters are as defined in Section 3.4, each over 10^6 jets. Every row is measured against the same target of 40 725 501 constituents, which is the target with the held-out probe removed. The kinematic channels are scored over the pooled constituents; the multiplicity m is one value per jet rather than one per constituent and is therefore not included in SW_1 .

the decoding back to physical units are all constructs that synthesise, and the design assembled from the first two blocks emits an encoded jet on every clock cycle at 360 MHz.

As a generative model for stress-testing, the sampler is adequate but not precise. It improves on an untrained prior by a factor 3.6 to 7.2 across the three channels and by 3.6 on SW_1 , while remaining 16 to 36 times above the floor set by the finite-sample metric estimator (Table 1). The bulk of each marginal is reproduced, and the pairwise densities of Figure 2 follow the broad shape of the target without resolving its finer structure. This generator would thus provide a downstream system like the Level-1 Scouting with realistic-looking objects, but it would not substitute for a physics-based simulation.

What limits that fidelity differs by channel. Firstly, integrating the same trained grid from the same noise using floating-point arithmetic reduces the W_1 -distance to the target by a factor 4.7 on η and 7.5 on ϕ , to 7.1 and 4.8 times the floor. The fixed-point implementation, not the learnt field, is therefore what bounds the accuracy of η and ϕ . This makes the quantisation widths the cheapest improvement available; they were set to conifer’s defaults and never tuned (Section 3). Widening them does not require changing the model’s architecture, and at 20.4% device occupancy there is room for wider words.

Secondly, for p_T the diagnosis is the opposite. The floating-point model already sits 10 times above the floor, and quantisation adds only a further factor 1.6 on top. The discrepancy is thus a property of the model itself, not due to quantisation. This is evident in Figure 1, as the generated distribution follows the low- p_T peak but cuts off at 163 GeV, whereas the training data extends to 1055.25 GeV.

An objection is that this is an artefact of the finite sample size rather than a failure of the model: the tail is below 10^{-5} probability density, so a random sample of 10^6 jets could contain zero high- p_T jets by chance. The target dataset disproves this: of its 29 057 185 jets, 1781 lie above 163 GeV, which is a fraction of $f = 6.13 \times 10^{-5}$. If the model reproduced the target, the number of jets above the cut-off in a sample of size N would be binomially distributed with mean Nf , giving 61.3 expected jets at $N = 10^6$. The probability of drawing none is

$$(1 - f)^N \approx 2.4 \times 10^{-27}.$$

The empty tail is thus not an unluckily sampled set. This is confirmed empirically by repeating the generation with 29×10^6 samples, matching the size of the training set: Figure D.1 in Appendix D shows the tail remains unpopulated.

The cut-off is therefore a property of the model directly. The heavy right tail carries little probability mass so adds little to a squared-error regression loss in Eq. (9). Reweighting the training data towards high p_T is the natural fix and is explored in Appendix D.

The model reported here is that it is the model that was translated to firmware, which had to be fixed before a hyperparameter scan could be completed. That scan, reported in Appendix C,

suggests a different optimum, so the two do not coincide and the scores above can be improved. Adopting that optimum is not simply a matter of reading off the best score, as the model is bound by the chip resources. Whether a configuration fits on the device cannot be predicted a priori, as the HLS estimates are, as the name suggests, estimates. Acquiring a reliable number requires a full build. Model selection is therefore the balance of most accurate configuration that still fits, which is confirmed once it has been built. The pipeline that produced the numbers above is, however, now in place: training and translation are automated end to end, so applying the result of Appendix C is a matter of rerunning the written scripts.

The same applies to the circular encoding of ϕ , which was not implemented on model-size grounds. However, encoding $\phi \mapsto (\sin \phi, \cos \phi)$ visibly reduces fluctuations in the ϕ marginal at the tree depth used here (Appendix A) thus should be investigated more. Implementation requires an additional channel in every field step and an atan2 lookup table in the decoding HLS block, increasing model size. The accuracy gain is moreover coupled to model size: at smaller tree depths the encoding introduces sub- 2π high-amplitude periodic behaviour in the sampled ϕ marginal, reducing the fidelity.

Finally, the choice of prior distribution p_0 is itself a free parameter that this work did not vary. The flow is only required to carry p_0 onto the data, so any p_0 closer to the target gives the field an easier path to follow. The ϕ channel is the clearest example of this, as it is uniform on $[-\pi, \pi)$ by symmetry. A matched uniform prior reduces the flow to the identity map there and removes the channel from the learning problem altogether, allowing for model size reduction. Appendix F shows this to work. Extending the idea to the remaining channels is less obvious, as it requires distributions that fit the characteristics of p_T and η while cheap enough to generate in firmware.

5.2 Hardware feasibility and throughput

Section 1 posed two obstacles to an on-the-fly generator for Level-1 Scouting: sustaining the 40 MHz collision rate for extended periods, and implementation on the trigger FPGAs rather than on external hardware. The build of Section 4.2 satisfies the second requirement immediately, and comes within reach of the first.

The design is fully pipelined with an initiation interval of one cycle at a clock period of 2.77 ns, producing 36×10^7 jets per second. This translates to a lower bound of 22.5×10^6 minimum-bias events per second, just over half the collision rate. Since the initiation interval is already one, further throughput from a single instance can only be bought by speeding up the clock. Alternatively, two instances on the same device would reach at least 45×10^6 cumulative events per second, more than the target throughput. At the 20.4% occupancy of Table 2 there is room for both at 40.8% of the chip, looking at resource counts alone. It remains to show the two jet streams can be merged into a single event stream, which additionally requires grouping a variable number of jets per event.

The noise sampled from the prior is currently supplied as external input; this part of the generator currently sits off the device. This is not an obstacle to construct the full end-to-end generator, as a Gaussian noise generator for this chip has already been implemented at 400 MHz [16], with external control over its seeding and its run length. It is thus suitable for our purpose.

Two independent effects nonetheless limit the amount of unique samples the model can produce. The first is the noise source: any on-chip generator is deterministic and has a finite cycle length, so for extended use of the generator, the period of the noise must be determined and compared against the number of samples the application consumes. If it falls short, a simple uniform generator supplying fresh seeds to the Gaussian source removes this bound at negligible cost. The second effect is the sampler itself, which outputs a finite set of representable values as it is working in fixed-point arithmetic. This means distinct noise inputs can be carried onto identical outputs, so the number of distinct jets the design can generate is finite regardless of the

noise it is fed. Larger word widths increases the number of representable outputs, which is the same adjustment already suggested in Section 5.1 to improve accuracy.

5.3 Constituent-level generation on FPGAs

The constituent-level model with hyperparameters identical to the jet-level model performed at lower accuracy than the jet-level model. Moreover, it was not built on an FPGA due to model size. Describing a jet by its $M = 16$ constituents increases the channel count from 3 to 49, and since the model consists of one BDT per channel per time step, the cost of a field step scales with channel count. Extrapolating Table 2 gives an estimated resource requirement of roughly 5.3 million LUTs, already exceeding the device by a factor three. It moreover is a lower bound for the reasons given in Section 4.3.

BUFF [4] proposes a solution using multi-output BDTs, each predicting several velocity components at once, which breaks the linear scaling of model size with the number of channels. Appendix E tests this architecture in Python, where it does not yet reproduce the target accurately, so tuning it into working form is the first step. Conifer, moreover, does not currently support the translation of multi-output BDTs to HLS C++. Extending conifer to emit multi-output ensembles is therefore a prerequisite for placing a constituent-level generator on a single trigger board.

6 Conclusion

This work set out to establish whether an on-the-fly generator of PUPPI kinematics can be built on the CMS Level-1 Trigger hardware itself, rybbubg fast enough to stress-test the Level-1 Scouting system at the 40 MHz collision rate. Combining BDT-based flow matching [4] with conifer [6], generators were trained on minimum-bias data at two levels of granularity: whole jets and the individual PUPPI particles constituting them. The jet-level model was successfully built on the `xcvu13p-flga2577-2-e` chip of the Phase-2 Level-1 Trigger. The training and translation pipelines are automated end to end and released as `puppibuff` [7].

The main result is the successful build on hardware. The velocity field, the AB(2) solver integrating it and the decoding back to physical units all synthesise correctly, and the design assembled from the first two occupies 20.4% of the FPGA. It achieves an output of one encoded jet per clock cycle at 360 MHz, or at least 22.5×10^6 minimum-bias events per second. This is half the collision rate from a single generator, and more than that when combining two instances, which the device has room for. The built design was verified to agree bitwise with its HLS simulation, so every stage of a flow-matching sampler can be expressed in trigger firmware and run there at the required throughput.

The fidelity of the jet-level sampler is adequate for stress-testing but not for physics. It improves on an untrained prior by a factor 3.6 to 7.2 per channel while remaining 16 to 36 times above the floor set by the finite-sample metric estimator, reproducing the bulk of each marginal and the broad shape of the pairwise densities without their finer structure. The limiting factor differs per channel: for η and ϕ the fixed-point precision implementation dominates, and for p_T the learnt field does, visible as a tail that cuts off at 163 GeV against the 1055.25 GeV of the training data.

The jet-level build should be read as a proof of concept for the generator that Level-1 Scouting actually requires, which is the constituent-level one. That model was trained and evaluated in Python only. Describing a jet by its $M = 16$ constituents raises the channel count from 3 to 49 and, since a field step instantiates one BDT per channel, pushes the extrapolated cost to roughly 5.3 million LUTs, some three times the capacity of the chip as a lower bound. The architecture

demonstrated at jet level therefore does not transfer to constituent level as it stands.

Several things remain before it can. The linear scaling of model size with channel count has to be broken, for which BUFF's multi-output BDTs are the natural candidate. They do not yet work in Python (Appendix E), so they must first be tuned into a functioning model. Additionally, conifer does not currently translate multi-output ensembles to HLS C++, so extending it is a further prerequisite for an on-chip constituent-level generator.

On the firmware side, several tasks are still open before the on-chip model is a self-sustaining end-to-end generator. Firstly, the quantisation widths for the HLS implementation were left at conifer's defaults and should be tuned to improve fidelity from the FPGA. Secondly, the prior is still expected as input so it needs an on-device Gaussian generator whose period is checked against the intended run length. Thirdly, the decoding block is written and synthesised but should be included in this build. Lastly, the separate jets still have to be merged into events with varying multiplicity.

On the model side, the p_T tail calls for reweighted training, a matched prior removes at least the ϕ channel from the learning problem entirely, and the hyperparameter optimum found after the design had to be frozen has yet to be adopted.

Importantly, none of these are open questions about whether the approach works. The generator runs on the trigger hardware approaching the collision rate, and what separates it from a usable one is a clear set of extensions to the current build.

Acknowledgements

I am grateful to Sioni Summers and Jaime León Holgado for their enthusiasm, support, and expertise throughout the summer, for advising me where I was stuck and allowing me the freedom to steer the project. I thank Chris Brown, Ioannis Xiotidis, and Dimitrios Danopoulos for their useful feedback during our group meetings. I also thank Jelle Aalbers for his help with my application to the programme.

And finally, to all the 2026 CERN summer students, among them Adam, Allan, Arta, Ayoub, Benja, Bilal, Carmen, Chris, Dhruv, Dušan, Emil, Farhan, Farid, Filip, Gideon, Jesse, Lucas, Mahim, Marko, Maša, Matilda, Odina, Paul, Sam, Sara, Sarah, Sean, Shahd, Sima, Sofia, Sonia, Sotiris, and Vincenzo, thank you for an unforgettable summer.

Appendices

A Circular encoding of ϕ

To respect the circular topology of the azimuthal angle ϕ and enforce its 2π -periodicity, it should be encoded as the pair $(\sin \phi, \cos \phi)$ and thereafter decoded using atan2 . The effect of this encoding strategy on a smaller jet-level model is presented in Figure A.1 and on the model used throughout this work in Figure A.2.

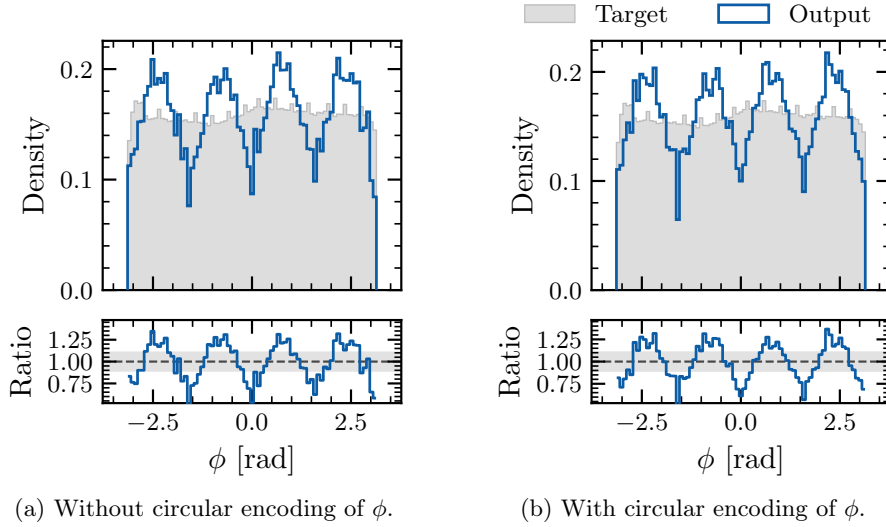


Figure A.1: Effect of encoding ϕ as $(\sin \phi, \cos \phi)$ for a small $d_{\max} = 3$ model.

With a small maximum depth $d_{\max}$, the model does not have enough capacity to keep ϕ on the unit circle. Sub- 2π periodic oscillating behaviour emerges when decoding by taking the argument of each point.

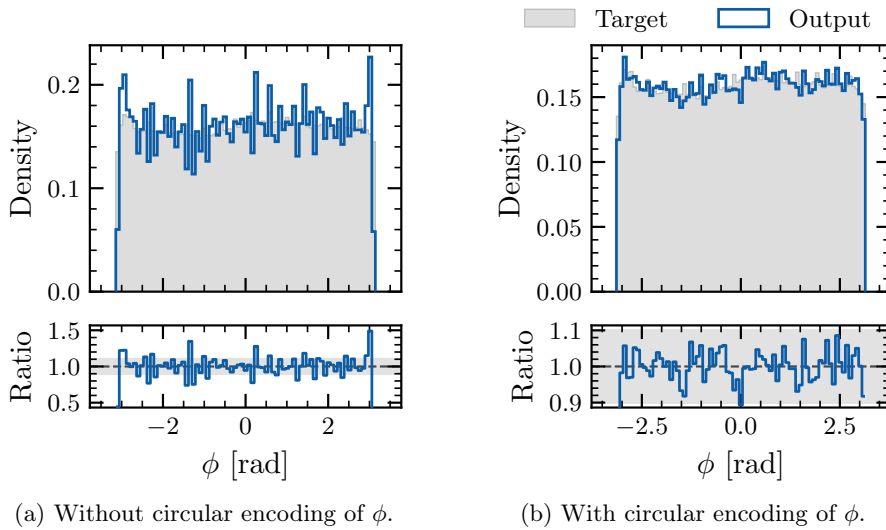


Figure A.2: Effect of encoding ϕ as $(\sin \phi, \cos \phi)$ for a larger $d_{\max} = 6$ model.

With a larger maximum depth $d_{\max}$, the encoding significantly reduces fluctuations in the output resulting in higher fidelity.

The circular encoding strategy was not adopted as it increases model size by adding a channel to the state vectors. Additionally, the decoding HLS block would need to include a lookup table for atan2 , increasing complexity from the previously used denormalisation.

Appendix F suggests a strategy that further improves the accuracy over the circular encoding, while decreasing model size.

B Numerical integrators

BUFF [4] proposes the use of a midpoint integration strategy with the BDT-based flow matching. We re-evaluated this choice while keeping in mind the end-goal of deployment on the FPGA.

Throughout this appendix, consider the initial value problem

$$\frac{dy}{dt} = f(t, y(t)), \quad y(t_0) = y_0,$$

integrated with a fixed step size h , writing $t_n = t_0 + nh$ and $y_n \approx y(t_n)$. Since a field evaluation is one full field step inference, each inference taking place on the FPGA when fully pipelining, the cost of a solver is set by the number of evaluations per step.

The (forward) Euler method is first order and uses one evaluation per step,

$$y_{n+1} = y_n + hf(t_n, y_n). \quad (\text{B.1})$$

The two-step Adams–Bashforth method (AB(2)) is second order, yet still costs one *new* evaluation per step, as the second slope is the one already computed on the previous step,

$$y_{n+1} = y_n + h \left(\frac{3}{2}f(t_n, y_n) - \frac{1}{2}f(t_{n-1}, y_{n-1}) \right), \quad (\text{B.2})$$

started with a single Euler step.

The midpoint rule is second order and requires two evaluations per step,

$$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}f(t_n, y_n)\right). \quad (\text{B.3})$$

Heun’s method is likewise second order with two evaluations per step,

$$k_1 = f(t_n, y_n), \quad k_2 = f(t_n + h, y_n + hk_1), \quad y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2), \quad (\text{B.4})$$

but calls the field a full step ahead rather than half a step, aligning the time steps with the time grid the BDTs were trained on.

We tested each solver with the same trained flow model and identical noise $\mathbf{x}_0$, over $n_{\text{steps}} = 9$ time steps. The following Figures B.1 to B.4 are the output results of this experiment.

Visually, and to be expected from the notes above, Heun achieves the best accuracy output. While it does not resolve the same level of detail, AB(2) achieves similar performance with half the required resource cost. Moreover, it outperforms the midpoint rule, again with half the required resource cost. This motivated the choice for AB(2) as the numerical integrator throughout the project.

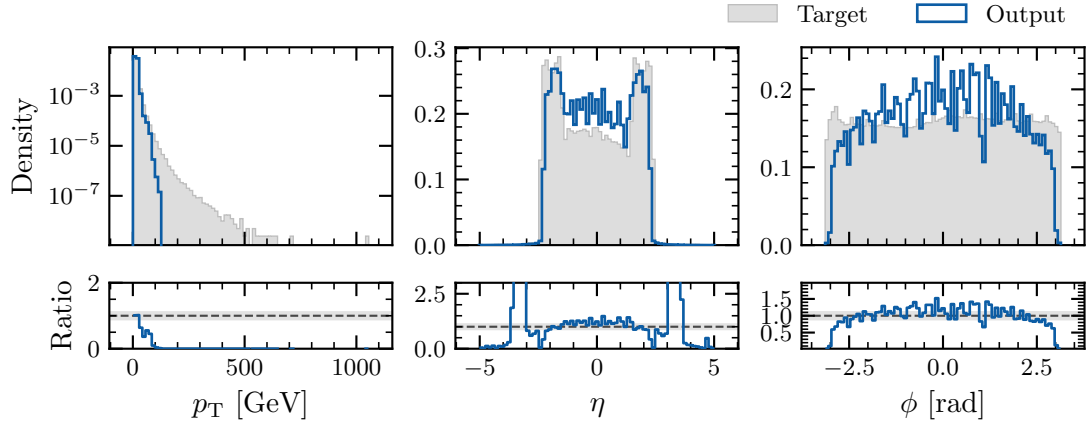


Figure B.1: Output distribution using the Euler integration method, Eq. (B.1).

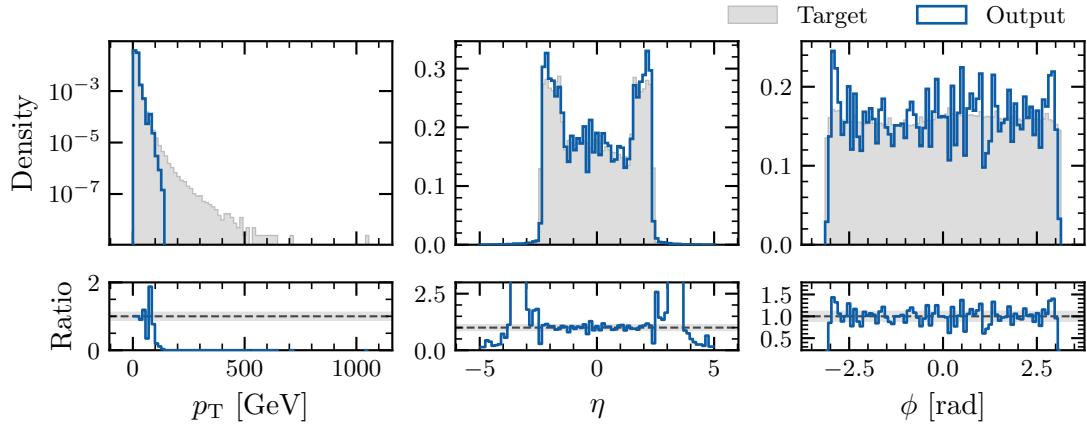


Figure B.2: Output distribution using the AB(2) integration method, Eq. (B.2).

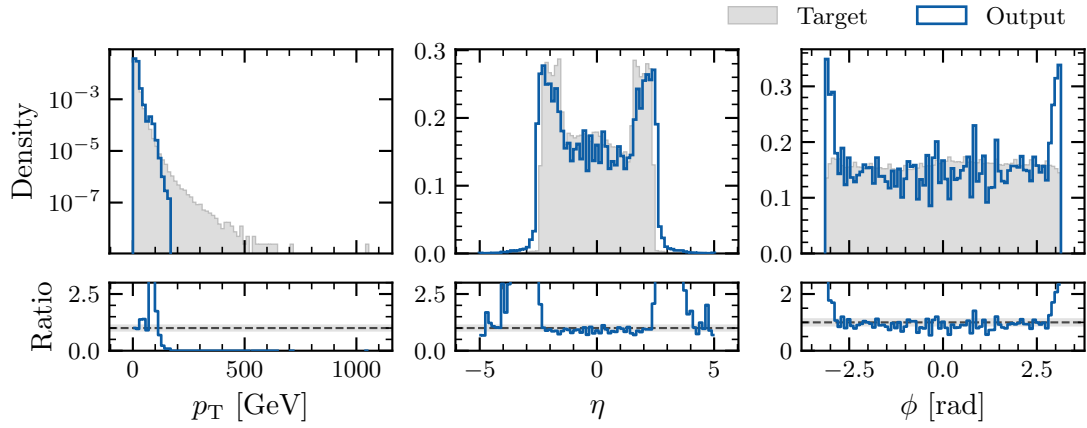


Figure B.3: Output distribution using the midpoint integration method, Eq. (B.3).

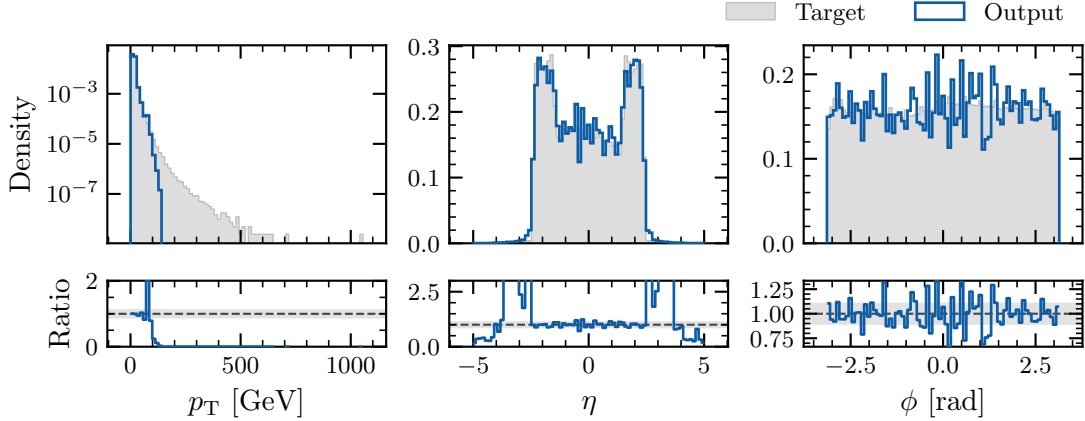


Figure B.4: Output distribution using the Heun integration method, Eq. (B.4).

C Hyperparameter scan

The model configuration of Section 3.2 had to be fixed before the scan reported here was complete, since synthesising and building a design takes considerably longer than training one. The scan covers the three hyperparameters that set the size of the velocity field,

$$d_{\max} \in \{3, 4, 5, 6\}, \quad n_{\text{trees}} \in \{50, 75, 100, 125\}, \quad n_{\text{steps}} \in \{10, 15\},$$

giving 32 grid points. Each is trained on the jet-level dataset with every remaining setting of Section 3.2 unchanged, sampled at 10^6 jets and ranked by SW_1 , the metric of Section 3.4 that measures inter-channel correlations. Table C.1 gives the ranking.

Accuracy alone does not provide the model to build because the resource cost of a configuration is not known in advance. The HLS estimates overshoot the final build by a factor 5.2 on this design (Section 4.2), so whether a given grid point fits on the `xcvu13p-flga2577-2-e` is only determined by a full build, which takes more time and effort than the training.

The leaf-count column nevertheless gives a proxy for model size. The grid holds one BDT per channel per time step, so the upper bound on its total leaf count scales with all three axes. The configuration carried to firmware, $d_{\max} = 6$, $n_{\text{trees}} = 50$ and $n_{\text{steps}} = 15$, is one of the 32 grid points but is not the best performing one.

$d_{\max}$	n_{trees}	n_{steps}	Leaves	$W_1(p_T)$	$W_1(\eta)$	$W_1(\phi)$	SW_1
6	100	15	2.88×10^5	0.148	0.00979	0.00699	0.00461
6	125	15	3.60×10^5	0.151	0.00945	0.00792	0.00514
4	100	15	7.20×10^4	0.152	0.0110	0.00701	0.00559
5	50	15	7.20×10^4	0.155	0.00943	0.00799	0.00560
5	75	15	1.08×10^5	0.149	0.00950	0.00804	0.00565
6	75	15	2.16×10^5	0.146	0.00976	0.00956	0.00578
4	125	15	9.00×10^4	0.149	0.00925	0.00823	0.00613
4	75	15	5.40×10^4	0.157	0.00972	0.00718	0.00622
6	50	15	1.44×10^5	0.148	0.00992	0.00716	0.00638
5	125	15	1.80×10^5	0.149	0.0114	0.00950	0.00644
5	100	15	1.44×10^5	0.151	0.0109	0.00978	0.00653
5	75	10	7.20×10^4	0.209	0.0151	0.0114	0.00658
3	125	15	4.50×10^4	0.161	0.00948	0.00814	0.00664
6	125	10	2.40×10^5	0.212	0.0144	0.0120	0.00726
3	75	15	2.70×10^4	0.167	0.00971	0.00820	0.00727
4	100	10	4.80×10^4	0.216	0.0128	0.0122	0.00744
4	50	15	3.60×10^4	0.168	0.00907	0.00793	0.00752
3	100	15	3.60×10^4	0.160	0.00971	0.00768	0.00753
6	75	10	1.44×10^5	0.214	0.0136	0.0140	0.00812
6	100	10	1.92×10^5	0.227	0.0135	0.0100	0.00832
3	125	10	3.00×10^4	0.223	0.0149	0.0118	0.00834
5	125	10	1.20×10^5	0.212	0.0161	0.0136	0.00834
4	75	10	3.60×10^4	0.219	0.0141	0.0128	0.00845
4	125	10	6.00×10^4	0.218	0.0131	0.0154	0.00848
6	50	10	9.60×10^4	0.216	0.0143	0.0139	0.00852
5	50	10	4.80×10^4	0.225	0.0128	0.0119	0.00903
3	50	15	1.80×10^4	0.180	0.0130	0.00812	0.00907
3	100	10	2.40×10^4	0.224	0.0145	0.0120	0.00916
4	50	10	2.40×10^4	0.211	0.0145	0.0158	0.00933
5	100	10	9.60×10^4	0.222	0.0134	0.0133	0.00935
3	75	10	1.80×10^4	0.231	0.0140	0.0129	0.00940
3	50	10	1.20×10^4	0.236	0.0137	0.0155	0.0121

Table C.1: Hyperparameter scan of the jet-level generator, all 32 grid points ranked by SW_1 . Every point is trained with the remaining settings of Section 3.2 unchanged and scored on 10^6 sampled jets. Leaves is the upper bound on the total leaf count of the grid, $n_{\text{steps}} \times n_{\text{channels}} \times n_{\text{trees}} \times 2^{d_{\max}}$, acting as a proxy for the full build size. The configuration carried to firmware is set in bold.

D The high- p_T tail

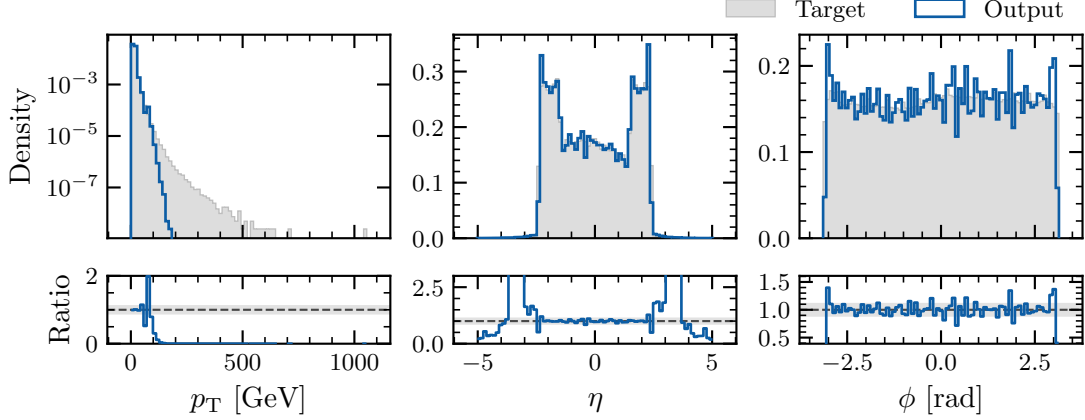


Figure D.1: Distribution of 29×10^6 samples from the implemented model. The high- p_T tail remains unpopulated.

Section 5.1 shows the jet-level model reproducing the low- p_T peak but cutting off at 163 GeV. Figure D.1 repeats the generation with 29×10^6 samples to match the size of the training set: the tail remains unpopulated, confirming that the cut-off is a model shortcoming and not due to sample size.

Since the tail carries little probability mass, and thus contributes little to the squared-error loss of Eq. (9), a typical solution is to reweight the training data towards high p_T . We give each jet a weight with weighting parameter α ,

$$w_i \propto (p_T)_i^\alpha, \quad (\text{D.1})$$

normalised to unit at the median and saturated above the 99.9% quantile of the training p_T so that individual extreme high- p_T jets cannot dominate the fit. $\alpha = 0$ recovers unweighted training with weight 1 across the set.

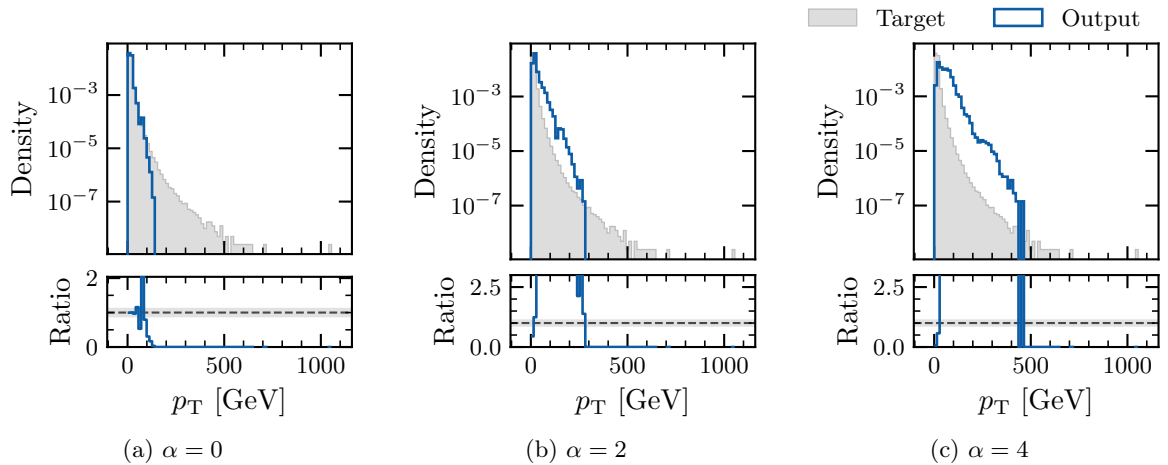


Figure D.2: Output marginal distributions of p_T after applying weighting strategy in Eq. (D.1) for different values of α .

Figure D.2 shows the generated p_T marginal with weighting for $\alpha \in \{0, 2, 4\}$. Although the cut-off moves upwards, the output distribution is lifted above the target across the whole range

between the peak and the cut-off. The longer tail is thus achieved by sacrificing fidelity on the bulk.

E Multi-output field models

Section 5.3 identified the scaling of the number of BDTs with the number of channels as the obstacle to placing a constituent-level generator on an FPGA. The grid consists of one regressor per channel per time step, so describing a jet by its constituents rather than as a whole multiplies the cost of a field step by 49/3. BUFF [4] avoids this by using multi-output trees, where a single BDT predicts several velocity components at once. While this would increase the required size of each BDT, it would allow the number of BDTs to be significantly reduced. This appendix reports initial training of such a model on the constituent-level dataset in Python.

The channels are grouped by kinematic feature, three BDTs predicting the velocity of the p_T channels, the η channels and the ϕ channels, and a fourth BDT for the scalar multiplicity m . A field step is thus 4 BDTs rather than 49.

The multi-output field was trained with identical hyperparameters as used for the reported jet-level and constituent-level models.

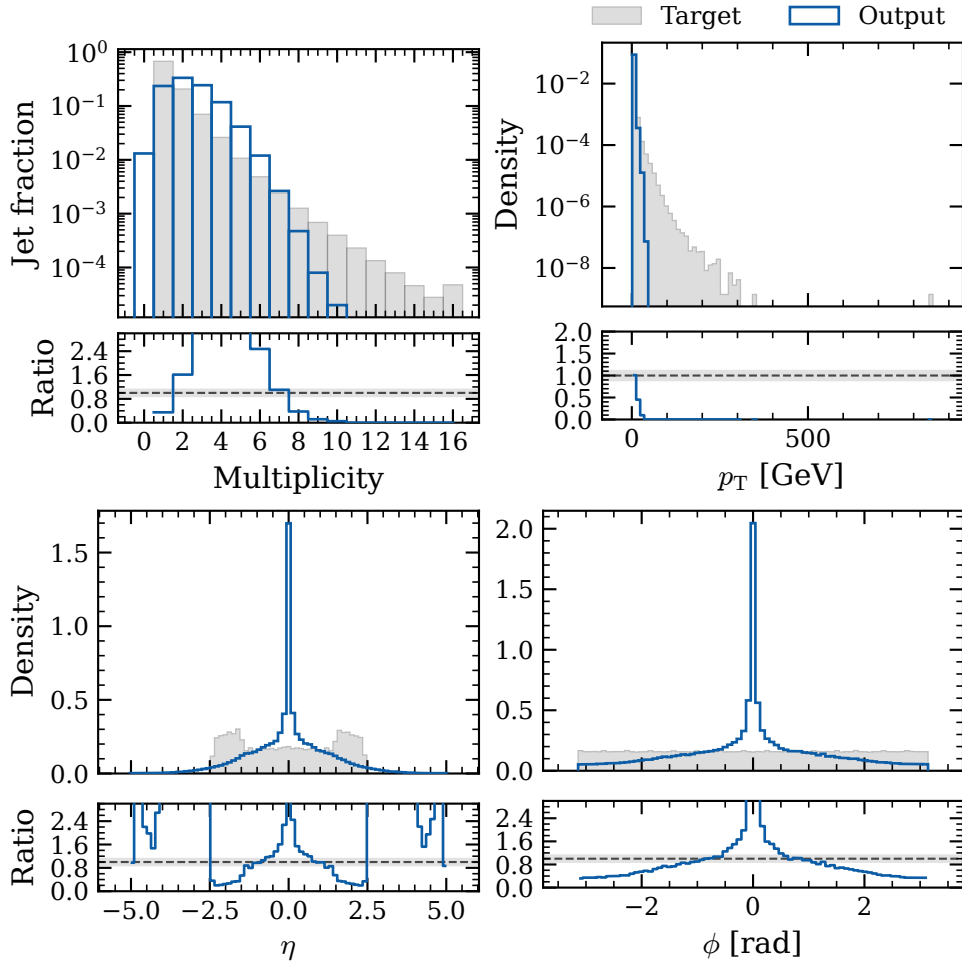


Figure E.1: Multiplicity distribution and marginal distributions of the pooled constituents in 10^6 jets sampled from the multi-output model with same hyperparameters as used throughout this work.

Figure E.1 shows the resulting marginals, and the output is worse than that of the single-output constituent-level model of Figure 5 on every channel. In particular, the output peaks sharply around the origin for η and ϕ , failing even to reproduce the general target shape. The p_T and multiplicity distributions are similarly shifted towards the origin, with the p_T cutoff moving below 100 GeV.

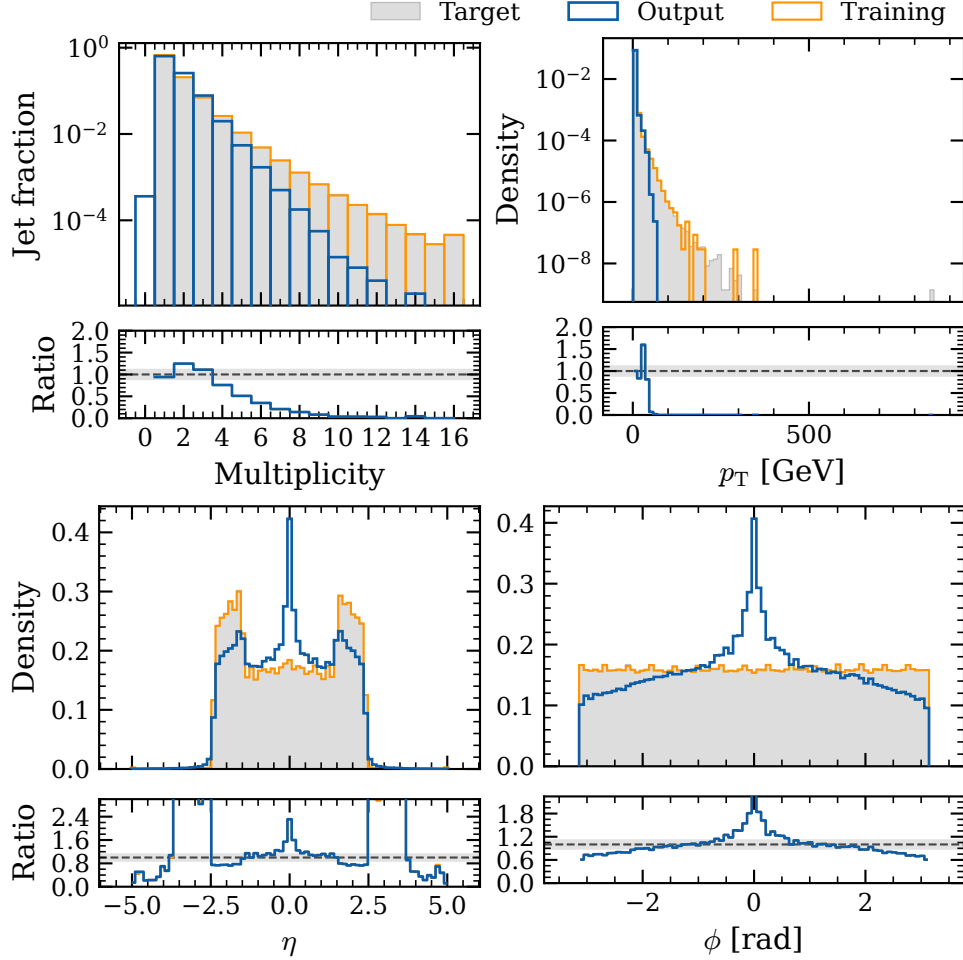


Figure E.2: Multiplicity distribution and marginal distributions of the pooled constituents in 10^6 jets sampled from the larger multi-output model.

Scaling up the model to 150 trees of maximum depth 10 on a time grid of $n_{\text{steps}} = 30$ shows a similar but dampened effect, displayed in Figure E.2. Now, the peak appears in η on top of a shape comparable to the target. The p_T marginal still cuts off below 100 GeV.

A possible explanation of this bias towards the origin is the zero-padding of Section 3.1. Namely, a padded constituent is placed at the origin in encoded space and decodes to $\eta \approx \phi \approx 0$, and since the multiplicity peaks at one or two constituents, most of the 16 slots are padding in a typical jet. A multi-output BDT shares one split structure across all slots of its channel, so those splits are fit predominantly to the padding. The single-output model of Figure 5 uncouples each slot by giving each a separate BDT, and this shows the same peak far less prominently. Replacing the zero-padding with uniform noise or draws from the target marginal may remedy this symptom, but establishing the cause is the work remaining before the architecture can be adopted.

F Choice of prior distribution

The choice of prior $p_0 = \mathcal{N}(0, 1)$ in Section 2.2 was motivated by ease of generation and the fact the input events get normalised. However, by choosing a noise distribution that is closer to the target distribution, we may simplify the model’s task.

As an example of this, we can leverage the rotational symmetry along the beam axis to argue ϕ must be distributed uniformly on $[-\pi, \pi)$. Taking the standardised distribution $\mathcal{U}[-\sqrt{3}, \sqrt{3}]$ as the prior for the ϕ channel makes p_0 and p_1 coincide in encoded space.

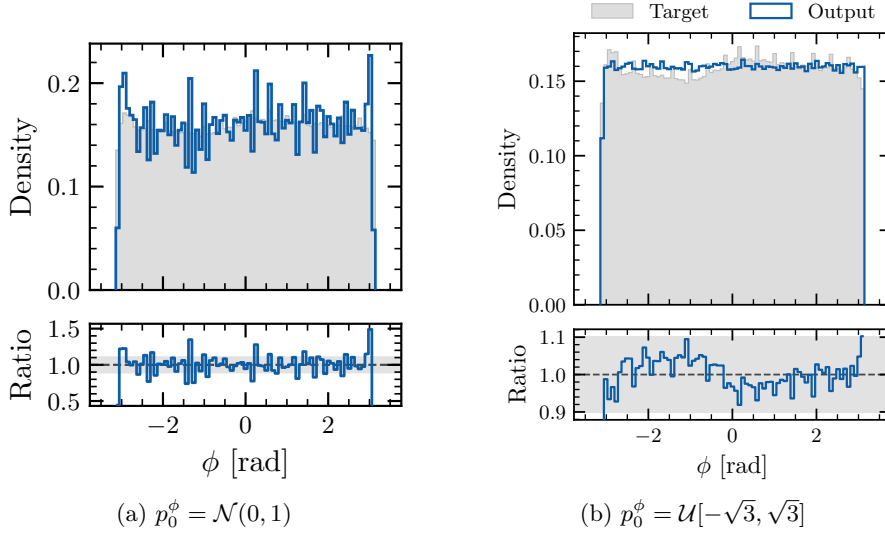


Figure F.1: Effect of changing the prior distribution for the ϕ channel from Gaussian to uniform.

Since the two endpoints now agree in distribution, we may pair in the conditioning step by drawing $\mathbf{x}_0^\phi = \mathbf{x}_1^\phi$ during training. The conditional probability path of Eq. (4) is then constant along ϕ and its target velocity vanishes. As this holds true for all paths along ϕ , this becomes easy for the model to learn. Then, at generation time, the prior for ϕ is passed through unchanged, making ϕ uniformly distributed by construction. Figure F.1 confirms this visually.

The major gain is in the model size rather than in its accuracy. The velocity field consists of one BDT per channel per time step, but a channel whose field is known to vanish a priori needs no trees at all. This would potentially allow the removal of a third of the ensemble from the jet-level design along with its resource cost. The prior itself also becomes cheaper, as uniform noise is less expensive to generate in firmware than the Gaussian one of Section 5.2.

The proposed construction relies on knowing the target marginal in closed form, which is what the symmetry supplies for ϕ alone. It also treats ϕ as independent of the other channels, which holds for the marginal of a single jet but not necessarily for constituents within one. Moreover, extending the idea to p_T and η requires distributions that are both closer to those targets and cheap enough to sample in firmware, and no candidates were identified here.

Bibliography

- [1] Gilbert Badaro et al. “40 MHz Level-1 Trigger Scouting for CMS”. In: *EPJ Web of Conferences* 245 (2020). Ed. by C. Doglioni et al., p. 01032. DOI: 10.1051/epjconf/202024501032.
- [2] Rocco Ardino and the CMS collaboration. “Development and demonstration of the CMS Phase-2 Level-1 trigger Data Scouting baseline system for HL-LHC”. In: *Journal of Instrumentation* 21.01 (Jan. 2026), p. C01024. DOI: 10.1088/1748-0221/21/01/C01024.
- [3] Francesco Vaselli et al. “End-to-end simulation of particle physics events with Flow Matching and generator Oversampling”. In: *Machine Learning: Science and Technology* 5.3 (Sept. 2024). arXiv:2402.13684 [hep-ex], p. 035007. DOI: 10.1088/2632-2153/ad563c.
- [4] Cheng Jiang, Sitian Qian, and Huilin Qu. *BUFF: Boosted Decision Tree based Ultra-Fast Flow matching*. arXiv:2404.18219 [physics.ins-det]. Apr. 2024. DOI: 10.48550/arXiv.2404.18219.
- [5] Yaron Lipman et al. *Flow Matching for Generative Modeling*. arXiv:2210.02747 [cs.LG]. Feb. 2023. DOI: 10.48550/arXiv.2210.02747.
- [6] Sioni Summers et al. “Fast inference of Boosted Decision Trees in FPGAs for particle physics”. In: *Journal of Instrumentation* 15.05 (May 2020). arXiv:2002.02534 [physics.comp-ph], P05026. DOI: 10.1088/1748-0221/15/05/p05026.
- [7] Koen Waggeveld. *puppibuff: BDT-based flow matching for on-the-fly event generation on FPGA firmware*. Version 1.0. 2026.
- [8] Oliver Brüning and Markus Zerlauth. *LHC operation and the High-Luminosity LHC upgrade project*. arXiv:2505.03535 [physics.acc-ph]. May 2025. DOI: 10.48550/arXiv.2505.03535.
- [9] Daniele Bertolini et al. “Pileup Per Particle Identification”. In: *Journal of High Energy Physics* 2014.10 (Oct. 2014). arXiv:1407.6013 [hep-ph], p. 59. DOI: 10.1007/JHEP10(2014)059.
- [10] Matteo Cacciari, Gavin P. Salam, and Gregory Soyez. “FastJet user manual”. In: *The European Physical Journal C* 72.3 (Mar. 2012). arXiv:1111.6097 [hep-ph], p. 69. DOI: 10.1140/epjc/s10052-012-1896-2.
- [11] Peter Holderrieth and Ezra Erives. *An Introduction to Flow Matching and Diffusion Models*. arXiv:2506.02070 [cs.LG]. June 2025. DOI: 10.48550/arXiv.2506.02070.
- [12] AMD. *AMD UltraScale+ FPGAs Product Selection Guide*. Product Selection Guide XMP103. Version 2.8. AMD, May 21, 2026.
- [13] David F. Griffiths and Desmond J. Higham. “Linear Multistep Methods—I: Construction and Consistency”. In: *Numerical Methods for Ordinary Differential Equations*. Series Title: Springer Undergraduate Mathematics Series. London: Springer London, 2010, pp. 43–60. DOI: 10.1007/978-0-85729-148-6_4.
- [14] Sioni Summers. *conifer: Fast inference of Boosted Decision Trees in FPGAs*. Version v1.9. Version 1.9. CERN, July 6, 2026.

- [15] Sloan Nietert et al. *Statistical, Robustness, and Computational Guarantees for Sliced Wasserstein Distances*. arXiv:2210.09160 [stat.ML]. Oct. 2022. DOI: 10.48550/arXiv.2210.09160.
- [16] Umran Yungucu. *Gaussian Random Number Generator and Generative Adversarial Networks Implementation on FPGA for Generating CMS Events at 40MHz*. Oct. 2022.