1. Introduction

Aerosol Jet Printing (AJP) is a mask-less, direct-write additive-manufacturing technology capable of producing micro-scale conductive features on both planar and non-planar substrates. Because it can deposit a wide range of functional inks—including metals, polymers, and biomaterials—AJP has become a key enabler for flexible electronics, wearables, antenna fabrication, biosensors, and multilayer interconnects. The final performance of these devices is tightly linked to the morphology of the printed traces: variations in trace width, edge roughness, continuity, or overspray directly affect electrical resistance, mechanical reliability, and dimensional accuracy.

Work Flow of AJP illustrated in Figure 1

1. Atomization: The ink (liquid containing nanoparticles or functional molecules) is aerosolized—converted into a mist of tiny droplets—using either ultrasonic or pneumatic methods.
2. Aerosol Transport: A carrier gas (usually nitrogen) transports the aerosol to the print head.
3. Focusing Nozzle: A sheath gas surrounds and focuses the aerosol stream into a tight jet (as small as 10–100 micrometers wide).
4. Deposition: The focused jet is directed onto the substrate, which can be flat, curved, or irregular.

In practice, morphology is governed by a *high-dimensional* and *non-linear* interaction of process parameters such as atomiser voltage (ATM), sheath-to-carrier-gas focusing ratio (FR), carrier-gas flow rate (CGFR), and stage speed (SS).

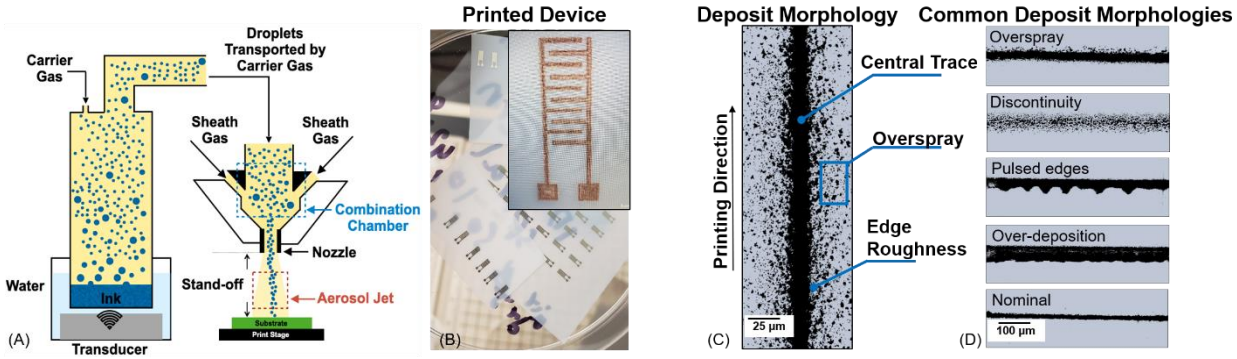

Figure 1: (A) Ultrasonic AJP workflow diagram. (B) Silver traces deposited on a flexible film (1 mm scale bar). (C) Single deposit highlighting key morphology metrics (25 µm scale bar). (D) Examples of typical AJP morphology classes (100 µm scale bar).

Recent breakthroughs in deep-learning—especially GAN (Generative Adversarial Network) a powerful, data-driven way to predict AJP outcomes. By adding process parameters as conditional inputs, a cGAN learns to translate a random noise vector and a parameter set into a high-resolution

morphology image, effectively standing in for the real print. Unlike straightforward regression, a cGAN embraces the natural randomness of droplet deposition and can produce many plausible variations for the same settings.

Yet AJP itself is highly complex. Key shape features—trace width (L_p), edge roughness (R_x), continuity (ψ), and overspray area (φ)—depend on a tangled, nonlinear mix of factors: atomizer voltage (ATM), carrier-gas flow rate (CGFR), focusing ratio (FR), stage speed (SS), nozzle–substrate gap, and ink rheology. Traditional tuning, based on design-of-experiments (DoE) trials and operator know-how, is

- Material-intensive – silver nano-inks cost > $400 USD per 100 mL
- Time-consuming – exploring a modest 5-factor, 3-level DoE exceeds 200 prints
- Local in scope – empirical maps rarely generalize beyond the explored window

A digital surrogate capable of predicting morphology a priori would dramatically accelerate process development, reduce ink waste, and enable closed-loop control. Physics-based computational fluid dynamics (CFD) or multiscale droplet-impact simulations can, in principle, predict morphologies, but they require high-fidelity nozzle flow models, contact-line dynamics, and ink rheology inputs—currently infeasible for fast design iteration. Recent breakthroughs in deep generative modelling, particularly Generative Adversarial Networks (GANs), provide a compelling data-driven alternative. By learning the underlying distribution of training images, a GAN can synthesize realistic images that capture subtle stochastic features beyond the reach of deterministic models.

1.1 Problem Statement:

We can create the AJP-based print according to the shape and the size given as the input. Even though it might look perfect in terms of the shape, **NO TWO PRINTS ARE IDENTICAL!** In terms of the physical Properties, the deposit morphology is not ideal and can vary from the intended design. This variation in the morphology can lead to differences in its properties (conductivity, voltage capacity etc.) and reduce the reliability in the process itself. The developed model contributes to the under-mentioned things in the domain of Direct Ink Printing:

- Reproducibility
- Move from Trial and Error based approach to Data Driven Approach.
- Increase Dataset balance and quality.

These contributions were made by our work through **CGAN (Conditional Generative Adversarial Network).**

A Conditional GAN (cGAN) extends this paradigm by incorporating control variables. In an AJP context, concatenating latent noise z with encoded process parameters c = {ATM, FR, CGFR, SS} allows the generator G(z|c) to output morphology images specific to a chosen parameter set. The discriminator D(x|c) assesses realism and parametric consistency, guiding G to honour physical causality as learned from data.

We present AJP-cGAN, the first conditional GAN trained on an extensive experimental dataset to predict AJP deposit morphologies. Our work makes the following contributions:

1. Comprehensive Image Dataset – 1950 traces were printed under 650 parameter combinations, imaged at 300×, binarized, cropped, and augmented to yield 23,400 labelled images (256 × 256 px).
2. Lightweight, Parameter-Aware Architecture – We designed a generator–discriminator pair with embedded parameter vectors that fits within a single consumer-GPU memory footprint while maintaining high resolution.
3. Multi-level Evaluation Framework – We introduce a dual-metric strategy combining AJP-specific morphological descriptors (Lw, ψ, Rq, φ) with canonical image similarity metrics (IoU, Precision, F1) for rigorous validation.
4. Insight into Process–Structure Links – By performing virtual parameter sweeps, we reveal how each process variable shapes morphology and identify regimes where the model diverges, informing targeted experimentation.

Dataset Description

2.1 Experimental Setup

- Ink: UTDAg40X silver nano-ink (40 wt.% concentration)
- Substrate: Glass microscope slides
- Nozzle: 30 G (233 ± 3 μm inner diameter)
- Printer: IDS NanoJet system
- Print Parameter Matrix: Four primary variables (defined below) were combinatorically varied to span an industrially relevant design space.
- Total Experiments: 650 unique parameter sets × 3 replicates = 1950 traces

2.2 Printing Parameters

The dataset used for training the cGAN was constructed from systematically varied process parameters in Aerosol Jet Printing (AJP), with each parameter discretized into a finite set of representative levels. As shown in Table 1, the four key parameters include Carrier Gas Flow Rate (CGFR), Focusing Ratio (FR), Atomizer Voltage (ATM), and Stage Speed (SS). These variables

were selected for their direct influence on aerosol transport dynamics, deposition resolution, and trace morphology. Due to the challenges associated with modeling continuous variables in GAN frameworks—such as instability and poor generalization in sparsely sampled regions—each parameter was treated as a categorical variable. Discrete levels were chosen based on experimentally validated process windows to ensure both physical feasibility and sufficient representation across the parameter space. This discrete encoding enabled robust conditional embedding within the GAN architecture and allowed the model to learn distinct morphology patterns associated with specific parameter combinations.

Table 1: Key AJP process parameters used as conditional inputs for the cGAN and their primary effects on deposit morphology

Parameter	Range / Levels	Functional Role	Variable
Carrier Gas Flow Rate (CGFR)	6, 9, 12 sccm	Controls the number of droplets transported in the aerosol stream.	CGFR
Focusing Ratio (FR)	1 – 15	Governs aerosol stream collimation, directly affecting jet diameter and trace width.	FR
Atomizer Voltage (ATM)	23.5, 30, 36.5 V	Influences droplet size and production rate, thereby modifying overspray and surface coverage.	ATM
Stage Speed (SS)	0.2, 0.4, 0.6, 0.8, 1 mm s^{-1}	Controls the effective material deposition rate, impacting trace continuity and thickness.	SS

2.3 Image Acquisition and Preprocessing Pipeline

All morphology images were acquired using an optical microscope at 300× magnification, resulting in native RGB images with a resolution of 2048 × 1536 pixels. To prepare the dataset for deep learning training, a structured preprocessing pipeline was implemented, as illustrated in Figure 3.

1. Otsu Thresholding was applied to convert high-resolution RGB micrographs into binary masks, segmenting the printed trace from the background. This unsupervised, histogram-based thresholding technique automatically identifies the optimal cutoff value to separate foreground (deposited material) from background, ensuring consistent and reproducible morphology extraction. Binary representation simplifies the learning task, focuses the model on structural features (e.g., line width, continuity, overspray), and reduces computational complexity during training.

2. Symmetric Cropping was performed to isolate the central region containing the trace morphology. Each image was cropped to a size of 1024 × 512 pixels, removing uninformative background and ensuring consistent framing across the dataset.
3. Data Augmentation was introduced to improve generalization and model robustness. Each cropped image was transformed using 180° rotation and horizontal/vertical flips, resulting in 12 augmented variants per original image. This augmentation preserved the geometric and morphological integrity of the deposit while increasing sample diversity across the training set.
4. Final Resize was applied to convert each image to a grayscale 512 × 512-pixel format, compatible with GPU memory constraints and optimized for convolutional network input. The grayscale representation retains all relevant morphological features while significantly reducing the input dimensionality.

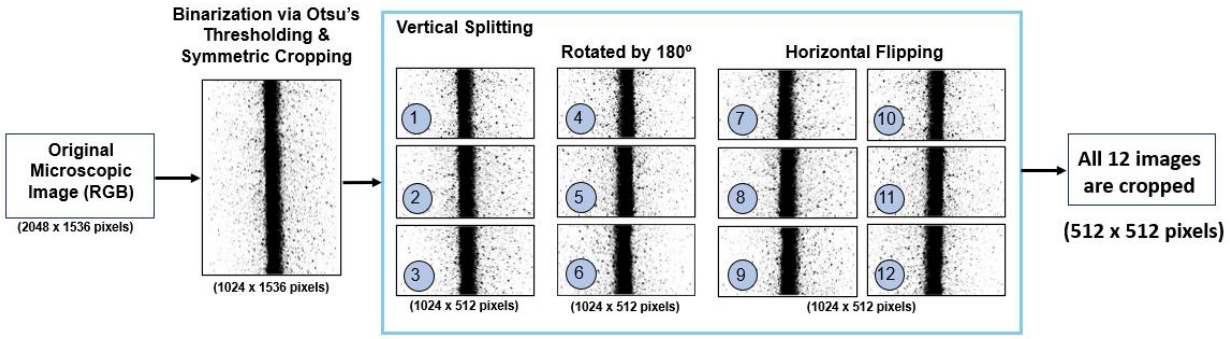

Figure 3: Overview of the image preprocessing and augmentation pipeline applied to microscope-captured AJP morphologies.

2.4 Final Dataset Statistics

The final dataset consisted of 23,400 grayscale binary images, derived from 1,950 unique experimental samples and their 12-fold augmented versions. Each image was labeled with a 4-dimensional vector representing the corresponding process parameters: Atomizer Voltage (ATM), Focusing Ratio (FR), Carrier Gas Flow Rate (CGFR), and Stage Speed (SS). These labels were discretized into categorical levels as described in Section 2.3 and encoded for conditional generation.

3. Conditional Generative Adversarial Network (cGAN) Formulation

This report details the development of a Conditional Generative Adversarial Network (cGAN) architecture tailored for predicting morphology outcomes in Aerosol Jet Printing (AJP). The framework was designed to synthesize morphology masks conditioned on four key discrete process parameters: Atomizer Voltage (ATM), Carrier Gas Flow Rate (CGFR), Focusing Ratio (FR), and Stage Speed (SS). Aimed at enabling high-throughput virtual experimentation, the

cGAN model was carefully designed, implemented, and evaluated to ensure it accurately reflects parameter-to-structure relationships.

3.1 Model Architecture Design

3.1.1 Generator

The generator network maps a stochastic latent vector and a process condition vector $z \in \mathbb{R}^{100}$ to a 256×256 binary morphology mask. To effectively integrate categorical process parameters, we employed separate embedding functions for each label:

$$\mathcal{E}_{ATM}: c_{ATM} \rightarrow \mathbb{R}^{a_{ATM}}, \mathcal{E}_{CGFR}: c_{CGFR} \rightarrow \mathbb{R}^{a_{CGFR}}, \mathcal{E}_{FR}: c_{FR} \rightarrow \mathbb{R}^{a_{FR}}, \mathcal{E}_{SS}: c_{SS} \rightarrow \mathbb{R}^{a_{SS}}$$

These embeddings project each discrete input to a lower-dimensional latent space, where all embedded vectors are concatenated with z and passed through a fully connected layer to initiate the feature map. The initial tensor is reshaped into a 128×64×64 feature map, followed by three upsampling blocks using nearest neighbor upsampling and convolution layers to achieve a final resolution of 256×256. Batch normalization and LeakyReLU activations were used for stability. A final Tanh activation was applied to ensure outputs lie within [-1, 1], consistent with discriminator expectations.

3.1.2 Discriminator

The discriminator receives a 256×256 input image along with four process parameters. Each label is embedded using an nn.Embedding layer and reshaped into a 2D spatial map matching the input image dimensions. These are concatenated with the input grayscale image across the channel dimension to form a 5-channel input tensor.

The network uses a series of five convolutional blocks with stride=2 and padding=1 to downsample the input from 256×256 to 4×4. Each convolution layer increases the number of feature channels (16 →32→ 64 →128→ 256), followed by BatchNorm, LeakyReLU, and Dropout. A final flattening and fully connected layer reduce the features to a scalar, passed through a Sigmoid activation to produce a real/fake probability.

3.2 Integration of Generator and Discriminator

The cGAN training loop was designed such that the generator and discriminator depend on each other's outputs:

Discriminator update: Uses both real images and fake images generated by the current state of G.

Generator update: Uses discriminator's prediction on fake images to update G.

Shared hyperparameters (learning rate = 0.0002, $\beta_1 = 0.5$) and normalization schemes (scaling pixel values to [-1,1]) ensured consistency. This reciprocal design ensures adversarial training remains balanced, allowing the generator to iteratively improve its output realism. The final architecture is visualized in Figure 4:

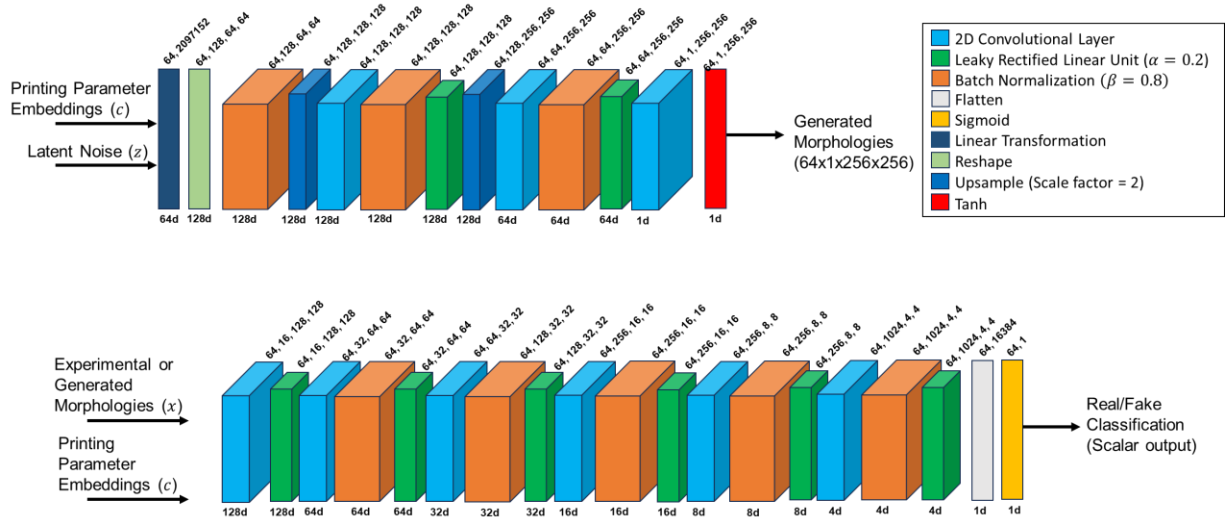

Figure 4: cGAN model architecture.

3.3 Model Optimization and Troubleshooting

Several challenges emerged during development:

- Mode collapse risk was mitigated using architectural stability (BatchNorm, Dropout).
- RGB image input increased training time and artifacts; we reverted to grayscale binary masks.
- Alignment issues in embedding shape and label broadcasting were resolved via consistent reshaping and concatenation.

We also ensured upsampling in G and downsampling in D were symmetrical (G: 64→128→256; D: 256→128→64→...→4), preserving resolution compatibility and feature correspondence.

3.4 Adversarial Training Dynamics and Computational Challenges

Training conditional GANs (cGANs) involves a delicate adversarial interplay between two neural networks—the generator (G) and the discriminator (D)—each with competing objectives. The generator aims to synthesize realistic morphology images that can deceive the discriminator, while the discriminator learns to distinguish real experimental images from generated ones. This zero-sum game is governed by the adversarial loss, where improvement in the generator's ability to "fool" the discriminator implies a simultaneous decline in the discriminator's performance, and vice versa.

Figure 5 shows the generator and discriminator loss curves over 100 training epochs. Initially, the generator loss increases as it struggles to produce convincing outputs, while the discriminator rapidly improves in detecting synthetic samples. Over time, both losses fluctuate, reflecting the dynamic equilibrium of adversarial learning. Notably, training instability—common in GANs—manifests in non-monotonic loss behavior, with the generator loss gradually rising after epoch 50. The best model, based on morphology fidelity and evaluation metrics, was identified near epoch 42. Beyond this point, signs of divergence suggest potential instability or overfitting.

Each epoch required approximately 5 hours and 30 minutes of GPU time, primarily due to the high-resolution 256×256 input images, batch size of 64, and complex embedding architecture. This prolonged training duration underscores the computational expense of morphology-aware GAN frameworks. Despite the model's ability to generate high-fidelity outputs, convergence remains fragile and demands careful hyperparameter tuning, regular monitoring, and potentially the use of advanced techniques such as gradient penalty, spectral normalization, or two-time scale updates to maintain training stability.

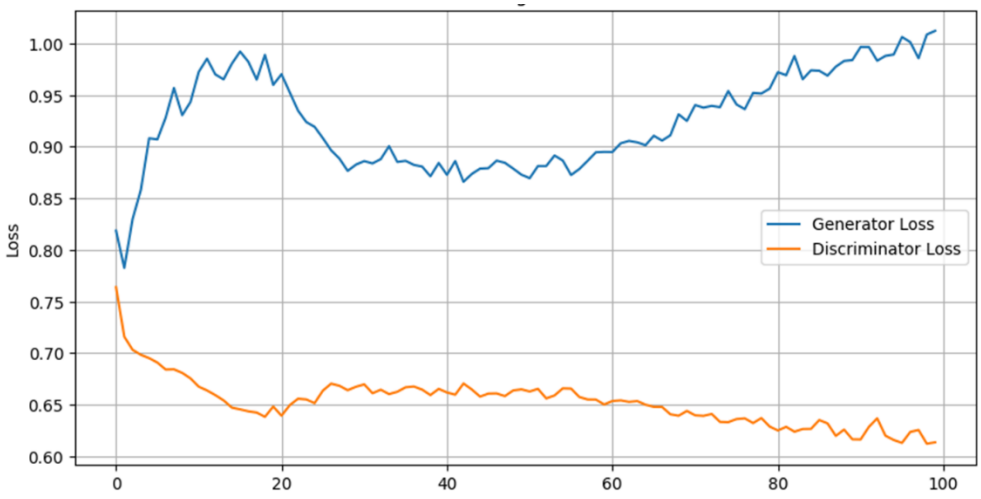

Figure 5: Generator and discriminator loss curves during training.

3.5 Performance Evaluation

Evaluating the performance of generative models is inherently challenging due to the lack of universal metrics that fully capture visual realism and structural fidelity. Each metric emphasizes different aspects of image quality, and may miss subtle yet important differences in feature continuity, roughness, or local artifacts. To address this, the performance of the cGAN was evaluated using both domain-specific morphological metrics and general image similarity indices.

Morphological attributes such as line width, material continuity, overspray, and edge roughness were extracted from binarized images using a custom OpenCV-based analysis pipeline (Figure 6).

These metrics provided insights into how well the generator mimicked true AJP-like deposition behavior across process conditions. The code for metric computation is publicly available on our GitHub repository. In parallel, image similarity metrics—including Jaccard Index, Precision, and F1 Score—were used to evaluate the spatial and structural consistency of generated images with respect to experimental data. These metrics consistently yielded values above 0.80, confirming strong pixel-level and morphological correspondence.

To support the statistical validity of the results, pairwise t-tests were conducted between experimental and generated metrics across all parameter combinations. No significant differences were observed at a 95% confidence level, underscoring the ability of the generator to produce morphologies that are statistically indistinguishable from real samples.

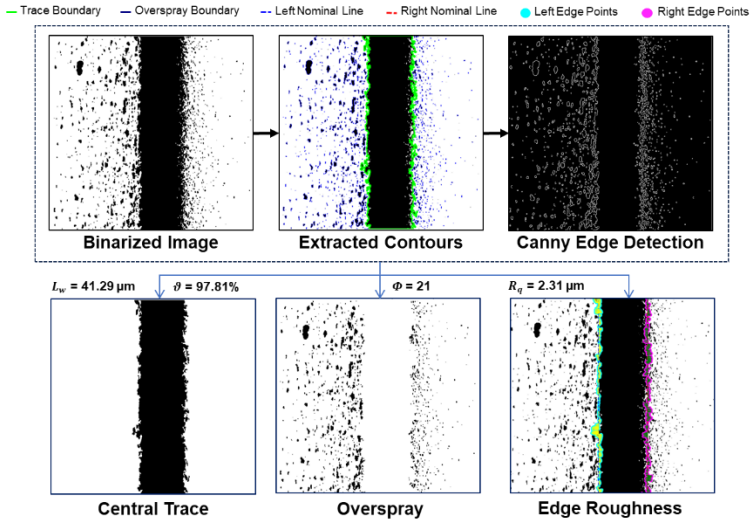

Figure 6: Workflow for extracting morphology metrics from generated images using OpenCV. Metrics include line width, overspray, edge roughness, and material continuity, enabling comprehensive evaluation of generative accuracy.

4. Results:

4.1 Morphology Generation with Conditional GAN

To evaluate the generative capability of the conditional GAN (cGAN) architecture, we selected a representative subset of 16 process parameter combinations covering a wide spectrum of line morphologies observed in the training dataset. This subset was chosen to test the model's ability to produce diverse output patterns when conditioned on known input parameters. Figure 7(A) presents a side-by-side visual comparison between experimentally acquired binary morphology images (left) and the corresponding cGAN-generated predictions (right). The generated morphologies qualitatively resemble the structural features of the experimental prints, including variations in line width, overspray behavior, and trace continuity.

To further assess performance, Figures 7(B–D) show kernel density estimations (KDEs) comparing the distribution of three morphological metrics—line width (Lw), overspray extent (Φ), and edge roughness (R_q)—between the experimental and generated image sets. Across all three metrics, the predicted distributions closely track the empirical distributions, capturing both the central tendency and spread of the data.

These results lead to several key observations:

(i) The cGAN demonstrates the ability to synthesize a broad range of morphology types, without collapsing to a narrow output space or favoring specific features;

(ii) The model shows sensitivity to distinct process parameter combinations, suggesting successful learning of the underlying parameter-to-structure mapping; and

(iii) The strong agreement in statistical distributions supports that the generator not only produces realistic images but also preserves the variability and structural statistics of the training dataset.

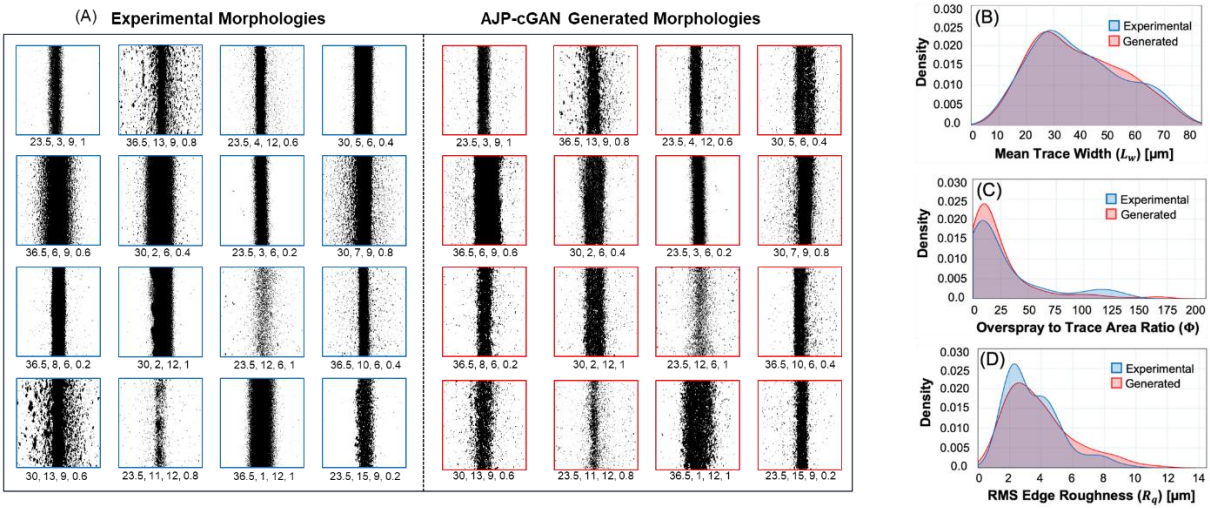

Figure 7: Comparison between experimental and cGAN-generated binary morphologies across 16 print conditions. (A) Visual comparison of real (left) and generated (right) images. (B–D) Kernel density estimates showing alignment of experimental and predicted distributions for line width (L_w), overspray (Φ), and edge roughness (R_q).

4.2 cGAN Evaluation Across Process Parameters

The conditional GAN (cGAN) model was evaluated across varying levels of four key process parameters to assess its ability to generate morphology patterns that align with experimental observations. For each parameter—Focusing Ratio (FR), Carrier Gas Flow Rate (CGFR), Stage Speed (SS), and Atomizer Voltage (ATM)—all other parameters were held constant to isolate the effect of the variable under study. Binary morphology masks were used for both experimental and generated data to enable direct visual and quantitative comparison. Figures 8-11 illustrate side-by-

side comparisons and statistical plots for key morphological descriptors, with corresponding image similarity metrics shown as column charts.

4.2.1 Effect of Focusing Ratio (FR)

The cGAN model demonstrated high sensitivity to changes in FR. As FR increased, the generated morphologies exhibited progressive narrowing of printed traces, accurately mirroring the trend observed in experimental data, as illustrated in Figure 8(A). This is reflected in the line width (L_w) measurements in Figure 8(C), where both experimental and generated values decreased from ~68 µm to ~36 µm across the tested FR range. The model also replicated the non-monotonic behavior of overspray (Φ)—initially decreasing with FR due to improved collimation Figure 8(E), but increasing at higher values, likely due to flow instability. While edge roughness (R_q) showed moderate variability, the model captured its downward trend, highlighting its capacity to encode both global and local structural features, shown in Figure 8(E). High similarity scores in Jaccard Index, Precision, and F1 Score above 0.7 in Figure 8(B) confirm the model's strong alignment with experimental data across FR conditions.

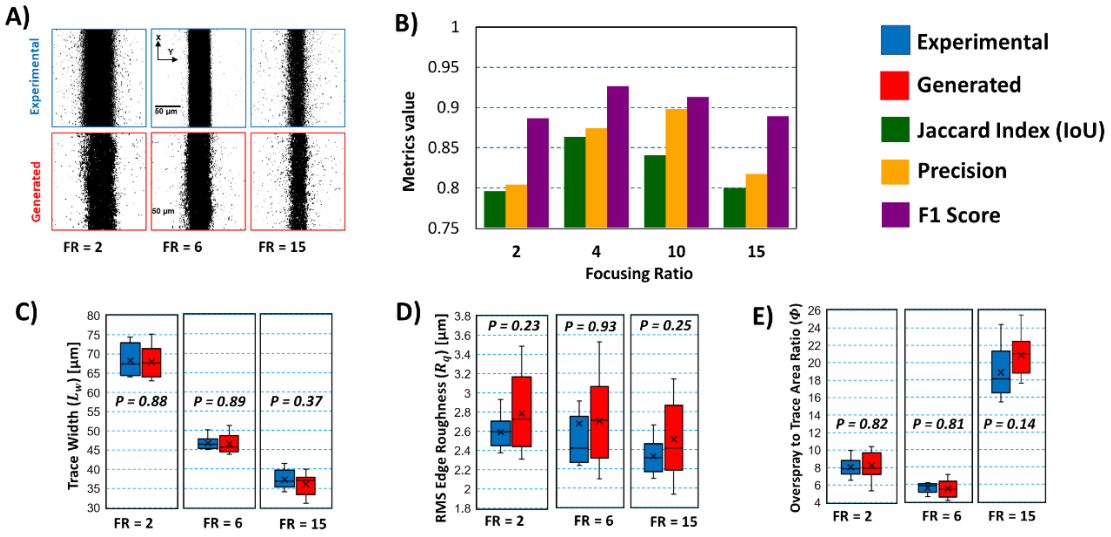

Figure 8: Comparison of experimental and cGAN-generated morphologies across varying FR, including (A) binary images, (B) image similarity metrics, and box plots for (C) line width (L_w), (D) edge roughness (R_q), and (E) overspray ratio (Φ).

4.2.2 Effect of Carrier Gas Flow Rate (CGFR)

The cGAN-generated outputs reflected the expected trend of increasing trace width with rising CGFR as illustrated in Figure 9(A), as more aerosolized droplets were transported into the deposition stream. Morphological metrics such as line width [Figure 9(B)] extracted from generated images were statistically indistinguishable from experimental counterparts across all CGFR levels. Overspray behavior also aligned well: Φ increased consistently [Figure 9(E)], a result of jet destabilization and secondary droplet formation at higher gas flow rates. The model

effectively represented changes in both overspray spread [Figure 9(E)] and droplet-induced edge degradation [Figure 9(D)], with high correspondence in R_q values. Image similarity scores in Figure 9(B) remained stable across the CGFR range, supporting the robustness of the generator in modeling mass transport-driven morphology changes.

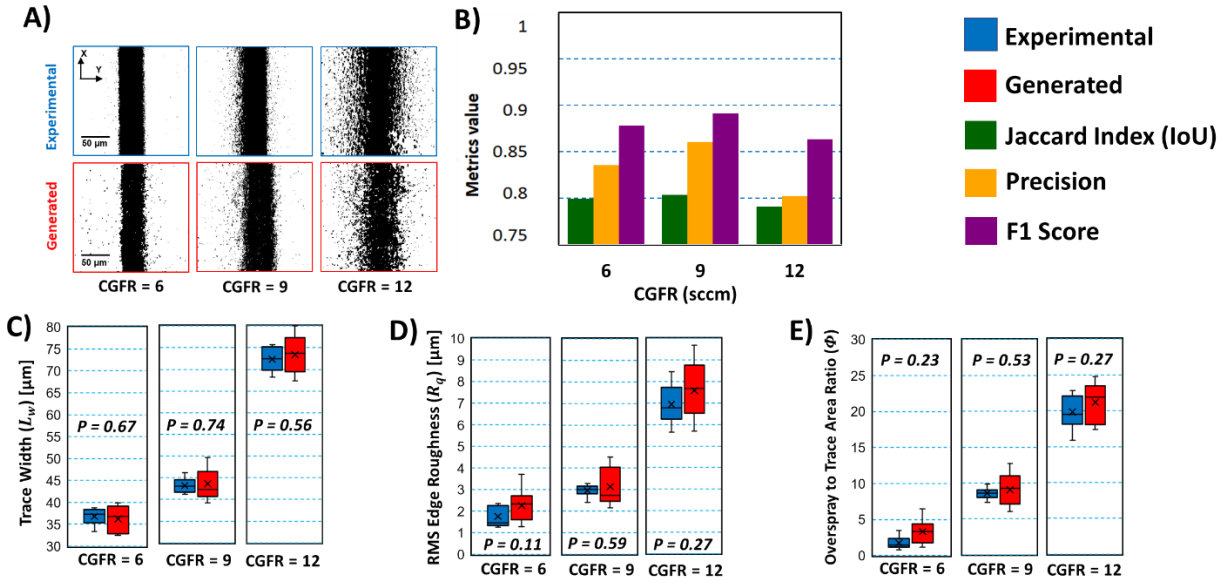

Figure 9: Comparison of experimental and cGAN-generated morphologies across varying CGFR, including (A) binary images, (B) image similarity metrics, and box plots for (C) line width (L_w), (D) edge roughness (R_q), and (E) overspray ratio (Φ).

4.2.3 Effect of Atomizer Voltage (ATM)

The cGAN was also evaluated across varying ATM values, which affect droplet size and atomization rate Figure 10(E). As voltage increased, generated traces became wider and more textured, aligning with the physical trend of increased material throughput, as shown in Figure 10(A). The model predicted both the broadening of trace width (L_w) [Figure 10(C)] and the corresponding degradation in edge smoothness (R_q) [Figure 10(D)]. The overspray droplet area (CA_{OS}) was also reasonably approximated, although spatial distributions showed slight deviations at higher voltages. Nevertheless, prediction accuracy remained high, with only minor reductions in similarity metrics in Figure 10(B) at extreme ATM levels, indicating that the generator retained its capability even under more complex atomization conditions.

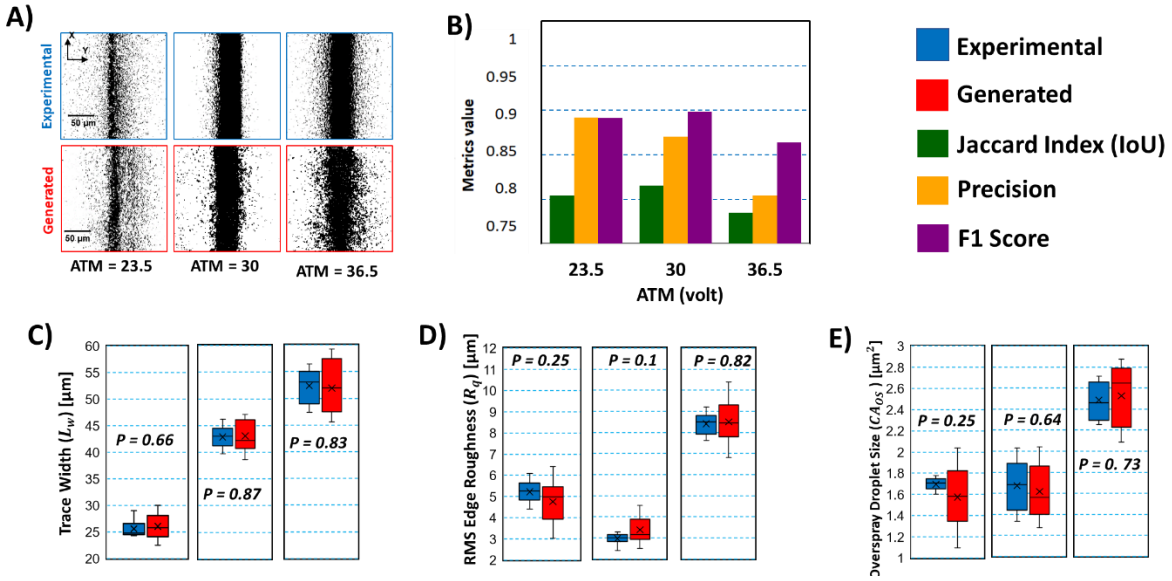

Figure 10: Comparison of experimental and cGAN-generated morphologies across varying ATM, including (A) binary images, (B) image similarity metrics, and box plots for (C) line width (L_w), (D) edge roughness (R_q), and (E) overspray ratio (Φ).

4.2.4 Effect of Stage Speed (SS)

Varying stage speed resulted in clear changes in generated morphology, particularly in material continuity and trace density. At lower speeds, generated traces were wide and continuous, while higher speeds led to thinner, fragmented lines, reflecting reduced droplet overlap, shown in Figure 11(A). The cGAN learned these spatial patterns and reproduced them with high fidelity. Continuity score (ϑ) in Figure 11(D) representing morphological segmentation, decreased with SS in both predicted and actual datasets. The model captured this trend well, suggesting that it effectively internalized the spatiotemporal relationship between stage velocity and trace formation in Figure 11(A & C). Across SS levels, similarity metrics in Figure 11(B) were consistently high, indicating precise prediction of both feature placement and structural transitions.

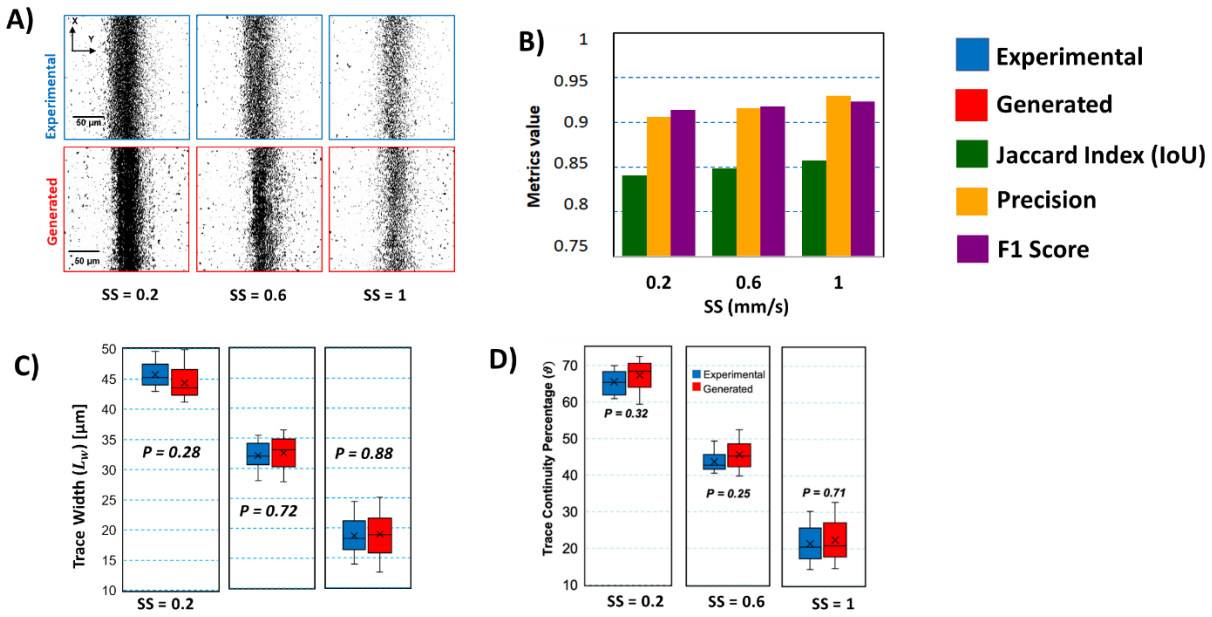

Figure 11: Comparison of experimental and cGAN-generated morphologies across varying SS, including (A) binary images, (B) image similarity metrics, and box plots for (C) line width (L_w), (D) trace continuity (ϑ).

4.3 cGAN prediction scenarios

Figure 12 illustrates three representative scenarios observed in the morphology prediction results generated by the residual conditional GAN model. In Scenario A, which accounts for approximately 80% of the cases, the model successfully generates deposit morphologies that closely match the corresponding experimental images in terms of line width, overspray distribution, and edge sharpness. This indicates effective learning of the underlying process-structure relationship and strong generalization across well-represented process conditions.

Scenario B, which accounts for approximately 18% of the generated outputs, reveals cases where the residual cGAN model over-predicts overspray and exhibits excessive black pixelation, deviating from the corresponding experimental observations. One key underlying issue in these cases may be partial mode collapse, a common failure mode in GANs where the generator starts producing similar or redundant outputs regardless of variations in the input noise vector z. This collapse limits the diversity of generated morphologies, leading to repetitive, artifact-laden patterns that fail to capture the full range of conditional variations — particularly when the training data is imbalanced or contains dominant classes such as high-overspray conditions. So, while the generator did not completely fail, it struggled to maintain diversity and realism under certain conditions, especially for high-overspray or underrepresented regions of the data distribution.

While the residual architecture itself offers advantages such as improved gradient flow and better preservation of spatial structure, it does not inherently prevent mode collapse. However, residual blocks can mitigate its severity by enabling deeper networks that learn more robust, fine-grained

features — which helps retain subtle conditional variations across samples. In this context, residual cGAN serves as a more stable foundation, but additional mechanisms are necessary to fully address mode collapse.

To mitigate the risk of mode collapse during training, several architectural choices were intentionally integrated into the model design. The use of residual blocks within the generator played a crucial role in stabilizing training by improving gradient flow and enabling the preservation of spatial features across deeper layers. This structural enhancement allowed the model to learn more robust mappings between latent vectors and morphology outputs, thereby reducing the likelihood of collapse into repetitive or oversimplified patterns. Additionally, conditional embeddings for process parameters (ATM, CR, FR, PS) helped ensure that the generator responded meaningfully to different input conditions, further supporting output diversity. While these choices provided a degree of protection against full mode collapse, the model did not incorporate more targeted strategies such as mini-batch discrimination, feature matching, or Wasserstein loss with gradient penalty (WGAN-GP) — all of which have been shown to promote sample diversity and reduce generative redundancy (Frogner et al., 2015). As such, although complete mode collapse was not observed, the presence of repetitive overspray artifacts in a subset of outputs (Scenario B) suggests that partial mode collapse may still have occurred under certain conditions (Park et al., 2020). Future iterations of the model may benefit from incorporating explicit diversity-promoting mechanisms to further enhance robustness and morphological variability.

Finally, Scenario C (~2–3%) encompasses cases where the model generates deposit morphologies for experimentally unviable or underrepresented process parameter combinations. Although the predicted patterns appear structurally coherent and physically plausible in isolation, they do not correspond to any observed experimental outcomes. This behavior highlights the generator's capacity to generalize beyond the training distribution, but also exposes a limitation: the model lacks an inherent understanding of physical feasibility (Saxena & Cao, 2024). In the absence of explicit constraints, the generator may produce morphologies that violate known process-material interactions, such as fluid flow dynamics, sintering behavior, or jet stability in AJP.

To address this, future work should explore the integration of physics-informed priors or hybrid modeling frameworks that embed domain knowledge directly into the generation process. For example, incorporating constraints derived from printability windows, material property limits, or simulation-derived feasibility maps can guide the model to produce only physically valid outputs. Additionally, introducing physically meaningful regularization terms or adopting hybrid models that combine data-driven learning with first-principles physics (e.g., differentiable simulation modules) can further improve the alignment between generated morphologies and real-world constraints. These enhancements would ensure that the model not only generates diverse and

realistic patterns but also respects the underlying physical limitations of the printing process.

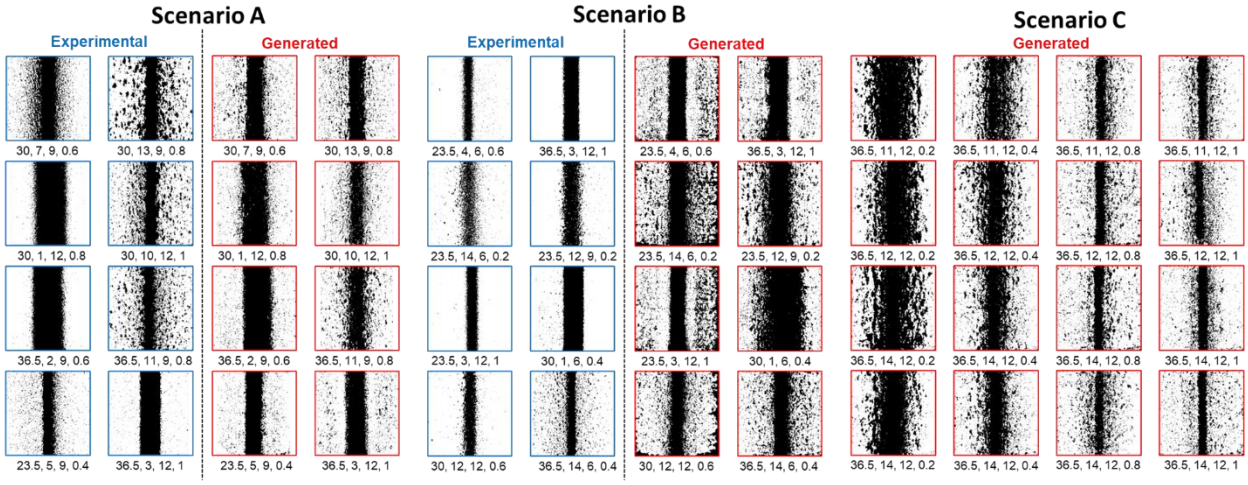

Figure 12: illustrates three prediction scenarios for AJP morphology generation.

5. Residual Conditional GAN Model

In conventional conditional GAN (cGAN) architectures, the generator learns to map a random noise vector and conditioning labels to realistic images. While effective in some cases, standard cGANs often suffer from training instability, vanishing gradients, and poor preservation of spatial structure — particularly when synthesizing high-resolution, fine-detailed images.

In our application — morphology generation for Aerosol Jet Printing (AJP) — these limitations are critical. Capturing precise features such as edge continuity, overspray, and line width requires stable learning and deep representations. Standard convolutional blocks in the generator struggle to maintain these fine-grained features over multiple upsampling layers.

To address this, we introduce a Residual Conditional GAN (Residual cGAN), where residual blocks are incorporated into the generator architecture (Xu et al., 2019). Each residual block learns a refinement over its input rather than a full transformation, allowing the network to reuse and preserve existing features. This design improves gradient flow, accelerates convergence, and enhances the quality and consistency of generated morphologies.

5.1 Residual Block Design

To address the limitations of standard convolutional blocks in deep generative networks, we incorporate residual blocks within the generator architecture (Park et al., 2022). In a standard block, the input undergoes a series of transformations represented as $F(x)$, and the output is entirely determined by this transformation. This structure lacks a mechanism to retain original feature information, which can lead to the degradation of critical spatial features and hinder effective gradient propagation, particularly in deeper networks (Khan et al., 2020). In contrast, a residual block introduces a skip connection that adds the original input x directly to the

transformed output $F(x)$, resulting in a final output of $F(x) + x$. This formulation allows the network to learn a residual mapping — that is, to focus on the incremental refinement needed rather than reconstructing the entire output from scratch. As illustrated in Figure 13, the residual block structure facilitates improved information flow across layers, enhances training stability, and preserves key morphological features such as edge continuity and overspray in generated AJP patterns (Esmaeili et al., 2023). The use of residual blocks is particularly beneficial in tasks requiring fine-grained detail and structural consistency, offering both better convergence and higher-quality image synthesis.

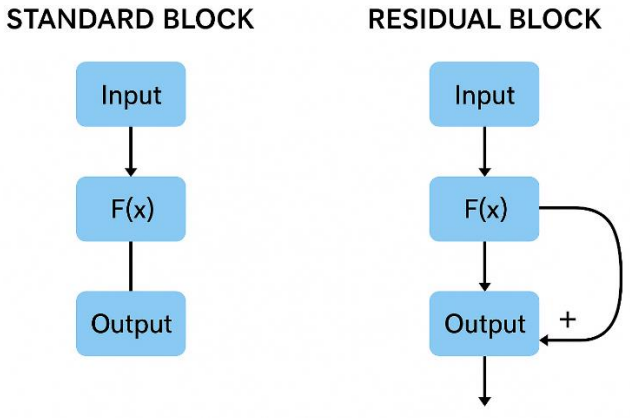

Figure 13: Architectural comparison between a standard block of cGAN (left) and a residual block (right).

5.2 Residual Architecture:

The Residual cGAN builds upon the standard conditional GAN architecture by integrating residual blocks into the generator network. The core idea is to improve training stability and feature preservation by enabling the network to learn residual functions instead of complete transformations. The model consists of a generator and a discriminator, both conditioned on process parameters: atomizer voltage (ATM), carrier gas rate (CR), flow rate (FR), and print speed (PS). The comparison of the cGAN Generator and Residual Block cGAN Generator is provided in Table 2.

Table 2: Architecture comparison of cgan generator and Residual Block cGAN Generator.

Component	Standard cGAN Generator	Residual Block cGAN Generator
Input	z + embedded labels	z + embedded labels
First Linear Layer	Linear → reshape to (B, 128, H, W)	Linear → reshape to (B, 128, H, W)
Upsampling Strategy	Upsample → Conv → BN → LeakyReLU (×2–3)	Upsample → Conv → BN → PReLU (×2), followed by residual refinement
Residual Block	Not present	Conv → BN → PReLU → Conv → BN + skip connection

Feature Flow	Sequential, each layer overwrites previous features	Additive refinement; original features preserved and enhanced
Depth Support	Limited by vanishing gradients	Deeper networks supported via improved gradient flow
Activation Function	LeakyReLU	PReLU (learnable)
Normalization	BatchNorm2d	BatchNorm2d
Output Layer	Conv2d (→ 1) → Tanh()	Conv2d (→ 1) → Tanh()
Final Output Shape	(B, 1, 256, 256)	(B, 1, 256, 256)
Training Stability	Moderate; sensitive to hyperparameters and learning rate	Improved; more stable due to residual connections

5.3 Residual cGAN Train Strategy:

The training strategy for the residual conditional GAN (cGAN) largely followed the standard cGAN setup, utilizing adversarial training with binary cross-entropy loss, conditional label embeddings, and the Adam optimizer with a learning rate of 0.0002 and betas (0.5, 0.999). Despite the deeper generator architecture introduced by residual blocks, no major changes were required in the training loop (Code available at GitHub repository). However, due to the increased model complexity, training required more computational resources, with each epoch taking approximately 3 hour and 20 minutes. The residual architecture contributed to more stable convergence and improved feature preservation, though care was taken to monitor for overfitting and maintain gradient stability through appropriate weight initialization and batch normalization.

6. Residual cGAN results:

Figure 14 illustrates the ability of the residual cGAN model to capture the morphological effects of individual AJP process parameters. By varying one parameter at a time while holding others constant, the model demonstrates sensitivity to key physical trends observed in experimental data. As SS increases, the generated deposit lines become progressively narrower and more uniform, reflecting the reduced material accumulation associated with faster substrate movement. Variations in flow rate FR result in visibly thicker lines and increased overspray, consistent with the higher volume of material delivered to the substrate. Changes in CGFR affect the focusing of the aerosol stream: lower CGFR produces sharper, more focused lines, while higher CGFR leads to broader and more dispersed morphologies. Finally, increasing ATM intensifies atomization and results in greater droplet dispersion, which the model captures through increasingly grainy and diffuse line structures. These outputs confirm that the residual cGAN successfully learns physically meaningful process-structure relationships and can serve as a valuable tool for simulating morphology evolution across a wide parameter space in AJP.

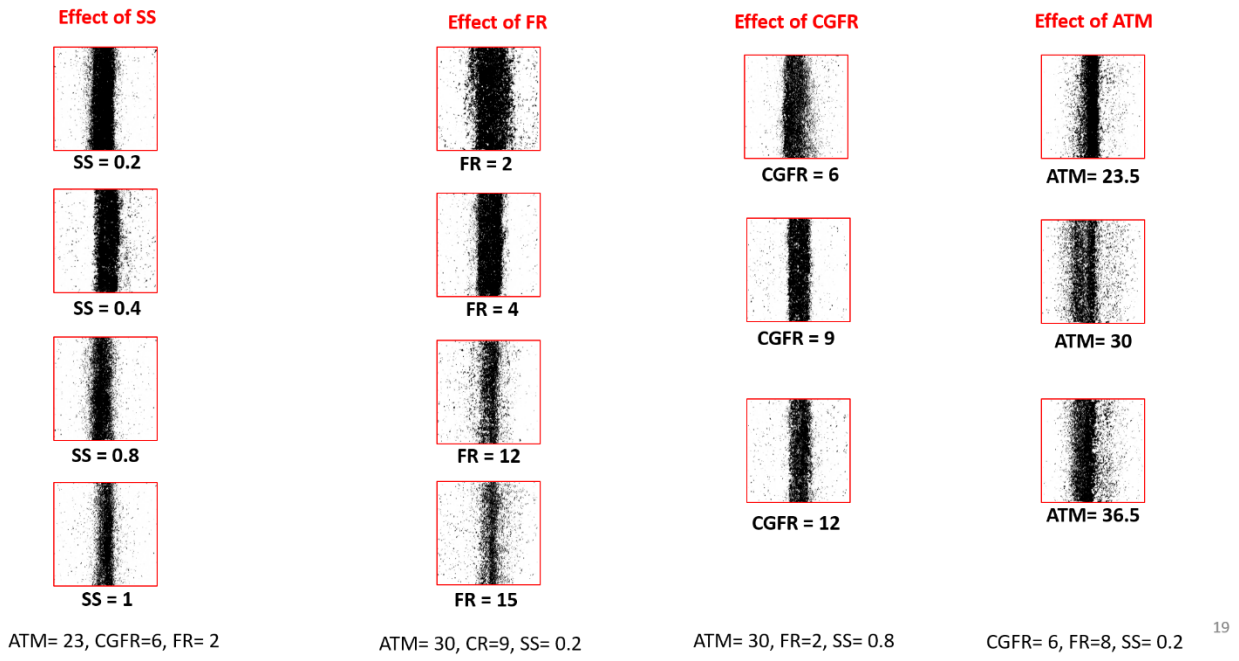

Figure 14: Residual cGAN-generated morphologies showing the effect of individual process parameters (SS, FR, CGFR, ATM). The model captures expected trends such as line narrowing with increased SS and overspray with higher FR, CGFR, and ATM, demonstrating its ability to learn process-structure relationships in AJP.

7. Challenges and Limitations

Despite the promising performance of the residual conditional GAN in capturing process-structure relationships in AJP morphology prediction, several technical and computational challenges were encountered during model development and experimentation.

First, due to time constraints and high training cost, the generated images were not subjected to comprehensive statistical, perceptual, or human-in-the-loop qualitative evaluation. While visual inspection confirmed morphological consistency across parameter variations, a more rigorous assessment—using metrics such as Fréchet Inception Distance (FID), Structural Similarity Index (SSIM), or downstream classification accuracy—remains to be performed in future work.

The training time per epoch was approximately 3 minutes and 20 seconds, which, when scaled to hundreds of epochs and parameter combinations, posed a significant computational bottleneck. Attempts to increase model expressiveness by incorporating RGB image channels (rather than binary grayscale inputs) further exacerbated memory usage and GPU load. Moreover, the RGB-based predictions appeared less smooth and more artifact-prone, likely due to increased complexity in learning three-channel representations without additional architectural modifications.

Additionally, initial experiments with StyleGAN-based architectures did not yield satisfactory results in this application. Although StyleGAN is known for high-fidelity image synthesis, it struggled to converge meaningfully on the AJP morphology data, potentially due to insufficient data volume, lack of continuous style-space correlation with process parameters, or incompatibility with discrete conditional labels.

Another significant challenge encountered was the integration of continuous-valued process parameters (e.g., stage speed, atomizer voltage, flow rates) into the generative framework. While conditional GANs are inherently designed to learn from discrete class labels, extending the conditioning mechanism to continuous variables requires architectural adaptations such as feature-wise modulation (FiLM), continuous embedding networks, or auxiliary regression-based objectives. These methods introduce additional complexity and instability in training, particularly when the mapping between continuous parameter values and morphology is nonlinear, non-monotonic, or sparsely sampled. To avoid these issues and ensure more stable conditioning, we discretized the process parameters into finite categorical levels based on known process windows and experimental resolution. While this approach facilitated robust embedding and convergence, it also introduced a degree of granularity loss in capturing fine variations across the process parameter space. Future work may address this limitation by implementing hybrid conditioning strategies that combine both discrete and continuous representations in a unified generative model.

These challenges underscore the trade-offs between model complexity, interpretability, and computational feasibility. They also highlight the need for future efforts focused on optimization of training efficiency, integration of physics-informed constraints, and hybrid generative approaches that balance realism with controllability and scalability.

8. Conclusion

In this work, we demonstrated the effectiveness of conditional Generative Adversarial Networks (cGANs), particularly a residual-enhanced cGAN architecture, in predicting deposit morphology in Aerosol Jet Printing (AJP) based on discrete process parameters. The residual cGAN exhibited improved visual fidelity, sharper edge definition, and better structural continuity compared to the standard cGAN, as observed through qualitative analysis. While a full quantitative evaluation is deferred due to computational constraints, the results indicate the model's strong potential in capturing meaningful process-structure relationships in printed morphologies.

We also identified three promising application scenarios for the trained residual cGAN model: (i) as an interactive visualization tool to support operators or decision algorithms in selecting suitable print parameters;

(ii) as a reference model within an in-situ monitoring system for detecting real-time deviations between predicted and actual deposit profiles; and

(iii) as a computationally efficient surrogate model to accelerate sequential design-of-experiment strategies such as Bayesian optimization.

Despite these promising outcomes, several challenges were noted. The integration of continuous conditioning variables proved unstable, prompting a shift to discretized parameter encoding. Moreover, the model faced limitations in training efficiency, generalization under sparse data, and prediction accuracy in experimentally unviable regions. Early trials using RGB images and StyleGAN architectures were also computationally intensive and yielded suboptimal results, while time constraints prevented rigorous statistical validation.

For future work, we recommend incorporating physics-informed constraints derived from governing principles of aerosol dynamics, droplet behavior, and substrate interactions to improve generalization and reduce artifact generation. Enhancing the model with temporal or signal-based data, such as real-time flow rates or droplet size distributions, may allow the capture of dynamic process effects. Expanding into three-dimensional morphology modeling using voxel grids or point cloud data, potentially with diffusion-based generative models, represents a promising direction for simulating volumetric structure evolution. Additionally, extending the output to include functional property predictions, such as electrical resistance or adhesion, will support a more comprehensive process–structure–function framework for intelligent, data-driven control in functional printing systems.

Reference:

Esmaeili, M., Abbasi-Moghadam, D., Sharifi, A., Tariq, A., & Li, Q. (2023). ResMorCNN model: hyperspectral images classification using residual-injection morphological features and 3DCNN layers. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, *17*, 219-243.

Frogner, C., Zhang, C., Mobahi, H., Araya, M., & Poggio, T. A. (2015). Learning with a Wasserstein loss. *Advances in neural information processing systems*, *28*.

Khan, A., Sohail, A., Zahoora, U., & Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial intelligence review*, *53*, 5455-5516.

Park, S. W., Huh, J. H., & Kim, J. C. (2020). BEGAN v3: avoiding mode collapse in GANs using variational inference. *Electronics*, *9*(4), 688.

Park, S., & Shin, Y. G. (2022). Generative residual block for image generation. *Applied Intelligence*, 1-10.

Saxena, D., & Cao, J. (2021). Generative adversarial networks (GANs) challenges, solutions, and future directions. *ACM Computing Surveys (CSUR)*, *54*(3), 1-42.

Xu, K., Cao, J., Xia, K., Yang, H., Zhu, J., Wu, C., ... & Qian, P. (2019). Multichannel residual conditional GAN-leveraged abdominal pseudo-CT generation via Dixon MR images. *IEEE Access*, *7*, 163823-163830.

Appendix:

CGAN

A.1 Environment & Hyper-parameters

```
python
CopyEdit
# A.1 - Runtime environment ----------------------------------------------------
import torch, random, numpy as np

SEED = 42
random.seed(SEED);  np.random.seed(SEED);  torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using", device, torch.cuda.get_device_name(0) if device.type == 'cuda' else "")

LATENT_DIM      = 100     # Length of noise vector z
IMG_SIZE        = 256     # Output resolution (square)
BATCH_SIZE      = 32
N_EPOCHS        = 200
LR_G            = 2e-4    # Learning rate – Generator
LR_D            = 2e-4    # Learning rate – Discriminator
BETAS           = (0.5, 0.999)
SAMPLE_INTERVAL = 1000    # steps between sampling grids
CHECKPOINT_DIR  = "checkpoints"
```

A.2 Dataset Loader

```
python
CopyEdit
# A.2 - Custom dataset for Aerosol Jet Printing (AJP) images ----------
import os
from PIL import Image
from torch.utils.data import Dataset

LABEL_MAPS = {
    'ATM' : {'23.5':0, '30.0':1, '36.5':2},          # 3 atomizer voltage classes
    'CR'  : {'6':0, '9':1, '12':2},                  # Carrier-gas flow rate
    'FR'  : {str(i):i-1 for i in range(1,16)},       # Focusing-ratio 1-15
    'PS'  : {'0.2':0,'0.4':1,'0.6':2,'0.8':3,'1':4}  # Stage speed mm s-1
}

def parse_filename(fname: str):
    """
    Parse 'ATM23.5_CR6_FR10_PS0.4_######.png'
    → dict(ATM=…, CR=…, FR=…, PS=…)
    """
    raw = fname.split('_')
    return {p[:3]: p[3:] for p in raw[:4]}

class AJPDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root = root_dir
        self.files = [f for f in os.listdir(root_dir) if f.endswith(".png")]
        self.transform = transform

    def __len__(self):
        return len(self.files)
```

```
    def __getitem__(self, idx):
        fname = self.files[idx]
        img   = Image.open(os.path.join(self.root, fname)).convert("L")
        if self.transform:   img = self.transform(img)

        labels = parse_filename(fname)
        y_ATM = LABEL_MAPS['ATM'][labels['ATM']]
        y_CR  = LABEL_MAPS['CR' ][labels['CR' ]]
        y_FR  = LABEL_MAPS['FR' ][labels['FR' ]]
        y_PS  = LABEL_MAPS['PS' ][labels['PS' ]]
        return img, (y_ATM, y_CR, y_FR, y_PS)
```

A.3 Conditional-GAN Architecture

A.3.1 Generator

python

CopyEdit

```
# A.3.1 - Conditional Generator ----------------------------------------------
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self,
                 latent_dim        = LATENT_DIM,
                 num_classes_ATM   = 3,
                 num_classes_CR    = 3,
                 num_classes_FR    = 15,
                 num_classes_PS    = 5,
                 img_size          = IMG_SIZE):
        super().__init__()
```

```
    # ⬇ label embeddings (one-hot → continuous)
    self.embed_ATM = nn.Embedding(num_classes_ATM, num_classes_ATM)
    self.embed_CR  = nn.Embedding(num_classes_CR , num_classes_CR )
    self.embed_FR  = nn.Embedding(num_classes_FR , num_classes_FR )
    self.embed_PS  = nn.Embedding(num_classes_PS , num_classes_PS )

    self.init_size = img_size // 4                # 256 → 64
    input_len      = latent_dim + sum([num_classes_ATM,
                                       num_classes_CR,
                                       num_classes_FR,
                                       num_classes_PS])

    self.l1 = nn.Sequential(
        nn.Linear(input_len, 128 * self.init_size ** 2)
    )

    self.conv_blocks = nn.Sequential(
        nn.BatchNorm2d(128),
        nn.Upsample(scale_factor=2),              # 64 → 128
        nn.Conv2d(128, 128, 3, 1, 1), nn.BatchNorm2d(128), nn.ReLU(True),
        nn.Upsample(scale_factor=2),              # 128 → 256
        nn.Conv2d(128,  64, 3, 1, 1), nn.BatchNorm2d(64 ), nn.ReLU(True),
        nn.Conv2d( 64,   1, 3, 1, 1), nn.Tanh()       # gray-scale
    )

def forward(self, noise, y_ATM, y_CR, y_FR, y_PS):
    # concatenate noise + embedded labels
    labels = torch.cat([
        self.embed_ATM(y_ATM),
        self.embed_CR (y_CR ),
```

```
            self.embed_FR (y_FR ),
            self.embed_PS (y_PS )
        ], dim=1)
        x = torch.cat([noise, labels], dim=1)
        out = self.l1(x).view(x.size(0), 128, self.init_size, self.init_size)
        return self.conv_blocks(out)
```

A.3.2 Discriminator

python

CopyEdit

```
# A.3.2 - Conditional Discriminator ----------------------------------------
class Discriminator(nn.Module):
    def __init__(self,
                 num_classes_ATM = 3,
                 num_classes_CR  = 3,
                 num_classes_FR  = 15,
                 num_classes_PS  = 5,
                 img_size        = IMG_SIZE):
        super().__init__()

        in_channels = 1
        self.label_embed = nn.Embedding(
            num_classes_ATM + num_classes_CR + num_classes_FR + num_classes_PS,
            img_size * img_size
        )

        def disc_block(in_f, out_f, bn=True):
            layers = [nn.Conv2d(in_f, out_f, 3, 2, 1), nn.LeakyReLU(0.2, True)]
            if bn: layers.append(nn.BatchNorm2d(out_f, 0.8))
            return layers
```

```
        self.model = nn.Sequential(
            *disc_block(in_channels + 1, 16, bn=False),   # +1 for label map
            *disc_block(16, 32),
            *disc_block(32, 64),
            *disc_block(64,128),
        )
        ds_size = img_size // 2 ** 4                    # 256/16 = 16
        self.adv_layer = nn.Sequential(
            nn.Linear(128 * ds_size ** 2, 1), nn.Sigmoid()
        )

    def forward(self, img, label_ids):
        # label_ids: concatenated integer codes [ATM, CR, FR, PS]
        lmap = self.label_embed(label_ids).view(img.size(0), 1,
                                        IMG_SIZE, IMG_SIZE)
        d_in = torch.cat([img, lmap], dim=1)
        out  = self.model(d_in).view(img.size(0), -1)
        return self.adv_layer(out)
```

A.4 Loss Functions & Optimizers

python

CopyEdit

```
criterion_GAN = nn.BCELoss()

generator     = Generator().to(device)
discriminator = Discriminator().to(device)

optimizer_G = torch.optim.Adam(generator.parameters()    , lr=LR_G, betas=BETAS)
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=LR_D, betas=BETAS)
```

A.5 Training Pipeline

python

CopyEdit

```
from torch.utils.data import DataLoader
from torchvision import transforms
from tqdm import tqdm
import os, itertools

# Data augmentation -- heavy overspray & translation jitter improve robustness
transform = transforms.Compose([
    transforms.RandomAffine(degrees=8, translate=(0.05,0.05), scale=(0.9,1.1)),
    transforms.ToTensor(),
    transforms.Normalize([0.5],[0.5])
])

dataset    = AJPDataset("data/ajp_aug256", transform)
train_loader = DataLoader(dataset, batch_size=BATCH_SIZE,
                          shuffle=True, num_workers=4, pin_memory=True)

valid_z = torch.randn(25, LATENT_DIM, device=device)

step = 0
for epoch in range(1, N_EPOCHS+1):
    loop = tqdm(train_loader, desc=f"Epoch {epoch}/{N_EPOCHS}", leave=False)
    for imgs, (yA,yC,yF,yP) in loop:
        imgs  = imgs.to(device)
        yA=yA.to(device); yC=yC.to(device); yF=yF.to(device); yP=yP.to(device)
        batch = imgs.size(0)

        # === Train Discriminator ===
```

```
optimizer_D.zero_grad()
real = torch.ones(batch,1, device=device)
fake = torch.zeros(batch,1, device=device)

real_out = discriminator(imgs,
                         torch.stack([yA,yC,yF,yP],1).long())
d_real  = criterion_GAN(real_out, real)

noise   = torch.randn(batch, LATENT_DIM, device=device)
gen_img = generator(noise, yA,yC,yF,yP).detach()
fake_out = discriminator(gen_img,
                         torch.stack([yA,yC,yF,yP],1).long())
d_fake  = criterion_GAN(fake_out, fake)

d_loss = (d_real + d_fake) / 2
d_loss.backward();  optimizer_D.step()

# === Train Generator ===
optimizer_G.zero_grad()
gen_img = generator(noise, yA,yC,yF,yP)
g_out   = discriminator(gen_img,
                        torch.stack([yA,yC,yF,yP],1).long())
g_loss  = criterion_GAN(g_out, real)
g_loss.backward();  optimizer_G.step()

loop.set_postfix(D_Loss=f"{d_loss:.3f}", G_Loss=f"{g_loss:.3f}")

# --- sampling & checkpointing ---
if step % SAMPLE_INTERVAL == 0:
    save_image_grid(gen_img[:25], f"samples/{step:07d}.png")
```

```
        step += 1
    torch.save(generator.state_dict(),
               os.path.join(CHECKPOINT_DIR, f"G_epoch{epoch:03d}.pth"))
```

A.6 Utility Functions

python

CopyEdit

```
import torchvision.utils as vutils
def save_image_grid(tensor, path, nrow=5):
    """
    Helper that rescales [-1,1] tensors to [0,1] and saves a grid PNG.
    """
    grid = vutils.make_grid((tensor + 1)/2, nrow=nrow)
    vutils.save_image(grid, path)
```

A.7 Inference Example

python

CopyEdit

```
# A.7 - Loading a trained Generator and synthesising a deposit ----------
state = torch.load("checkpoints/G_epoch150.pth", map_location=device)
G = Generator().to(device);  G.load_state_dict(state);  G.eval()

z    = torch.randn(4, LATENT_DIM, device=device)
y_ATM = torch.tensor([0,1,2,0], device=device)   # 23.5, 30, 36.5, 23.5 V
y_CR  = torch.tensor([0,0,1,2], device=device)   # 6,6,9,12 sccm
y_FR  = torch.tensor([4,7,10,13], device=device) # ratio 5,8,11,14
y_PS  = torch.tensor([2,2,1,0], device=device)   # 0.6,0.6,0.4,0.2 mm/s

with torch.no_grad():
    synth = G(z, y_ATM, y_CR, y_FR, y_PS)
```

```
save_image_grid(synth, "inference_example.png", nrow=2)
```

For Residual c-GAN

B-1 Runtime Environment & Global Hyper-parameters

python

CopyEdit

```
# B-1  –  Environment checks & reproducibility ------------------------------
import os, random, numpy as np, torch

SEED = 42
random.seed(SEED);  np.random.seed(SEED);  torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device, torch.cuda.get_device_name(0) if device.type=="cuda" else "")

# --- project-level constants (tuned in earlier experiments) ---
LATENT_DIM      = 100       # noise-vector length  z
IMG_SIZE        = 128       # 128×128 input/output for faster denoising cycles
BATCH_SIZE      = 16
EPOCHS          = 150
LR_G            = 2e-4      # generator - Adam
LR_D            = 2e-4      # discriminator - Adam
BETAS           = (0.5, 0.999)
CHECKPOINT_DIR  = "denoise_ckpts"
SAMPLE_EVERY    = 500       # steps per visual sanity-check
```

B-2 Residual U-Net-style Generator

python

CopyEdit

```
# B-2  –  Residual-Block helper ------------------------------------------
import torch.nn as nn

class ResidualBlock(nn.Module):
    """Two 3×3 convs + skip. Keeps spatial dims; boosts receptive field."""
    def __init__(self, ch: int):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(ch, ch, 3, 1, 1), nn.BatchNorm2d(ch), nn.PReLU(),
            nn.Conv2d(ch, ch, 3, 1, 1), nn.BatchNorm2d(ch)
        )
    def forward(self, x):
        return x + self.block(x)

# B-2.1  –  Conditional Generator with embedded process labels ----------
class Generator(nn.Module):
    """
    Inputs  : latent noise  z  + four categorical labels (ATM, CR, FR, PS)
    Outputs : 1-channel denoised AJP deposit (tanh-scaled to [-1,1])
    """
    def __init__(self,
                 latent_dim=LATENT_DIM, img_size=IMG_SIZE,
                 num_classes_ATM=3, num_classes_CR=3,
                 num_classes_FR=15, num_classes_PS=5):
        super().__init__()
        # — label embeddings (50-D each) —
        self.eATM = nn.Embedding(num_classes_ATM, 50)
        self.eCR  = nn.Embedding(num_classes_CR , 50)
```

```
    self.eFR  = nn.Embedding(num_classes_FR , 50)
    self.ePS  = nn.Embedding(num_classes_PS , 50)

    self.init_sz = img_size // 4                # 128 → 32
    fc_in        = latent_dim + 200              # 100 + 4×50
    self.fc      = nn.Sequential(
       nn.Linear(fc_in, 128 * self.init_sz ** 2),
       nn.LeakyReLU(0.2, inplace=True)
    )

    # — up-sampling path with residual refinement —
    self.up = nn.Sequential(
       nn.BatchNorm2d(128),
       nn.Upsample(scale_factor=2),             # 32 → 64
       nn.Conv2d(128,128,3,1,1), nn.BatchNorm2d(128), nn.PReLU(),
       ResidualBlock(128),

       nn.Upsample(scale_factor=2),             # 64 → 128
       nn.Conv2d(128,64,3,1,1),  nn.BatchNorm2d(64),  nn.PReLU(),
       ResidualBlock(64),

       nn.Conv2d(64,1,3,1,1), nn.Tanh()         # grayscale output
    )

def forward(self, z, yATM, yCR, yFR, yPS):
    labels = torch.cat([
       self.eATM(yATM), self.eCR(yCR),
       self.eFR(yFR),   self.ePS(yPS)
    ], dim=1)
    x = self.fc(torch.cat([z, labels], dim=1))
```

```
        x = x.view(x.size(0), 128, self.init_sz, self.init_sz)
        return self.up(x)
```

B-3 Patch-GAN Discriminator with Label Maps

python

CopyEdit

```
# B-3  –  Discriminator ----------------------------------------------------
class Discriminator(nn.Module):
    """
    A light Patch-GAN: classifies 16×16 patches as real / fake,
    conditioned on the same four categorical process labels.
    """
    def __init__(self, img_size=IMG_SIZE,
                 num_classes_ATM=3, num_classes_CR=3,
                 num_classes_FR=15, num_classes_PS=5):
        super().__init__()
        # flatten label IDs into a dense map that matches the image plane
        self.embed = nn.Embedding(
            num_classes_ATM + num_classes_CR + num_classes_FR + num_classes_PS,
            img_size * img_size
        )

        def block(in_f, out_f, use_bn=True):
            layers = [nn.Conv2d(in_f, out_f, 3, 2, 1), nn.LeakyReLU(0.2, True)]
            if use_bn: layers.append(nn.BatchNorm2d(out_f, 0.8))
            return layers

        self.model = nn.Sequential(
            *block(1+1, 16, False),  # +1 channel for label-map
            *block(16, 32), *block(32, 64), *block(64,128),
```

```
        )
        ds = img_size // 2**4                    # 8
        self.adv = nn.Sequential(nn.Linear(128*ds*ds, 1), nn.Sigmoid())

    def forward(self, img, yATM, yCR, yFR, yPS):
        ids  = torch.stack([yATM,yCR,yFR,yPS], 1).long()
        lmap = self.embed(ids).view(img.size(0),1,IMG_SIZE,IMG_SIZE)
        feat = self.model(torch.cat([img, lmap], 1)).view(img.size(0),-1)
        return self.adv(feat)
```

B-4 Loss Functions & Optimizers

python

CopyEdit

```
# B-4  –  GAN objective & Adam optimizers ----------------------------------
import torch.optim as optim
criterion = nn.BCELoss()

G = Generator().to(device)
D = Discriminator().to(device)

opt_G = optim.Adam(G.parameters(), lr=LR_G, betas=BETAS)
opt_D = optim.Adam(D.parameters(), lr=LR_D, betas=BETAS)
```

B-5 Training Loop (Joint Denoising + Adversarial)

python

CopyEdit

```
# B-5  –  One epoch of adversarial training ---------------------------------
from tqdm import tqdm

for epoch in range(EPOCHS):
```

```
    for imgs, yA, yC, yF, yP in tqdm(loader, desc=f"Epoch {epoch+1}"):
        imgs, yA, yC, yF, yP = [x.to(device) for x in (imgs,yA,yC,yF,yP)]
        b = imgs.size(0)
        valid, fake = torch.ones(b,1,device=device), torch.zeros(b,1,device=device)

        # ---- Train Discriminator ----
        opt_D.zero_grad()
        d_real = criterion(D(imgs, yA,yC,yF,yP), valid)

        z      = torch.randn(b, LATENT_DIM, device=device)
        gen_img = G(z, yA,yC,yF,yP).detach()
        d_fake  = criterion(D(gen_img, yA,yC,yF,yP), fake)

        d_loss = 0.5 * (d_real + d_fake)
        d_loss.backward();  opt_D.step()

        # ---- Train Generator ----
        opt_G.zero_grad()
        gen_img = G(z, yA,yC,yF,yP)
        g_loss  = criterion(D(gen_img, yA,yC,yF,yP), valid)
        g_loss.backward();  opt_G.step()
```

B-6 Sampling Utility for Progress Monitoring

python

CopyEdit

```
# B-6  –  Fixed-noise sampler saved every SAMPLE_EVERY steps ----------------
import torchvision.utils as vutils, os

def save_grid(tensor, path, nrow=5):
    """Rescale to [0,1] and save a PNG grid."""
```

```
    grid = vutils.make_grid((tensor+1)/2, nrow=nrow)
    vutils.save_image(grid, path)

if step % SAMPLE_EVERY == 0:
    with torch.no_grad():
        preview = G(fixed_z, fixed_ATM, fixed_CR, fixed_FR, fixed_PS)
    save_grid(preview, f"samples/{step:07d}.png")
```

Analyze_stripe_OSTU_all_metrics_X.ipynb

C-1 Environment & Global Settings

python

CopyEdit

```
# C-1  —  Imports and reproducibility switches  ------------------------------
import os, json, cv2, glob, random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

SEED = 42
random.seed(SEED);  np.random.seed(SEED)

plt.rcParams["figure.figsize"] = (6,4)
plt.rcParams["savefig.dpi"]    = 300
RESULTS_DIR   = "analysis_outputs"
os.makedirs(RESULTS_DIR, exist_ok=True)
```

C-2 Utility Functions

python

CopyEdit

```
# C-2.1 — Otsu thresholding wrapper ------------------------------------
def otsu_threshold(gray: np.ndarray):
    """
    Apply OpenCV's Otsu algorithm on a uint8 grayscale frame.
    Returns tuple (th_binary, th_val).
    """
    blur = cv2.GaussianBlur(gray, (5,5), 0)
    th_val, th = cv2.threshold(
        blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU
    )
    return th, th_val


# C-2.2 — Stripe mask & feature extractor --------------------------------
def stripe_metrics(th_img: np.ndarray):
    """
    From a binary mask of 'stripe' pixels:
      - width_mean, width_std
      - continuity (ratio of connected pixels along print path)
      - area coverage (% of image)
    Returns dict with metrics.
    """
    # skeletonize to extract 1-px centerline widths
    skel = cv2.ximgproc.thinning(th_img, thinningType=cv2.ximgproc.THINNING_ZHANGSUEN)
    # width per pixel via distance transform
    dmap = cv2.distanceTransform(255-th_img, cv2.DIST_L2, 5)
    widths = dmap[skel>0] * 2       # radius➔diameter
    labels, n = cv2.connectedComponents(skel)
    longest_seq = max(np.bincount(labels.flat)[1:], default=0)
```

```
    return {
        "width_mean" : widths.mean() if widths.size else np.nan,
        "width_std"  : widths.std(ddof=1) if widths.size else np.nan,
        "continuity" : longest_seq  / skel.size,
        "coverage"   : th_img.mean()
    }
```

C-3 Batch-wise Analysis Driver

```
python
CopyEdit
# C-3  —  Loop over every image in DATA_ROOT and aggregate metrics  ----------
DATA_ROOT = "stripe_dataset/validation256"
records   = []

for img_path in glob.glob(os.path.join(DATA_ROOT, "*.png")):
    fname = os.path.basename(img_path)
    gray  = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    th, thresh_val = otsu_threshold(gray)
    m = stripe_metrics(th)

    # parse filename to recover process parameters (shared helper from Appendix A)
    params = parse_filename(fname)          # returns dict ATM/CR/FR/PS
    records.append({**params,
                    "file"       : fname,
                    "otsu_val"   : thresh_val,
                    **m})

df = pd.DataFrame(records)
df.to_csv(os.path.join(RESULTS_DIR, "stripe_metrics.csv"), index=False)
print("Saved aggregated metrics →", RESULTS_DIR)
```

C-4 Exploratory Statistics & Plots

python

CopyEdit

```
# C-4.1  —  Summary stats per focusing-ratio (FR) ----------------------------
summary = (
   df.groupby("FR")
     .agg(width_mean=("width_mean","mean"),
          width_std =("width_std" ,"mean"),
          continuity=("continuity","mean"),
          coverage  =("coverage","mean"))
     .reset_index()
)
display(summary.style.format("{:.3f}"))

# C-4.2  —  Box-plot of stripe widths per stage-speed (PS) -------------------
import seaborn as sns
sns.boxplot(data=df, x="PS", y="width_mean")
plt.title("Distribution of Mean Stripe Width vs Stage Speed (PS)")
plt.xlabel("Stage Speed [mm s⁻¹]");  plt.ylabel("Mean Stripe Width [px]")
plt.tight_layout();  plt.savefig(os.path.join(RESULTS_DIR, "width_vs_speed.png"))
```

C-5 Threshold-Quality Confusion Matrix (Optional)

python

CopyEdit

```
# C-5  —  Simple confusion wrt manual ground-truth label in CSV  -------------
if os.path.exists("ground_truth_stripes.csv"):
   gt = pd.read_csv("ground_truth_stripes.csv")   # columns: file, label{good,bad}
   merged = df.merge(gt, on="file")
   merged["pred"] = np.where(merged["coverage"] > 0.10, "good", "bad")
```

```
    cm = pd.crosstab(merged["label"], merged["pred"])
    print(cm)
```

C-6 Persisting the Analysis Context

python

CopyEdit

```
# C-6  —  Save notebook settings & summary JSON for reproducibility ----------
config = {
    "seed"        : SEED,
    "data_root"   : DATA_ROOT,
    "n_images"    : len(df),
    "otsu_kernel" : 5,
    "metrics"     : list(df.columns)
}
with open(os.path.join(RESULTS_DIR, "analysis_config.json"), "w") as fp:
    json.dump(config, fp, indent=2)
```