

What a seed can—and cannot—remember

A mechanism-first essay for Phi9 · Public editorial candidate, 3 October 2026

A small number can summon a large picture. That is a striking experience, but it can hide where the work happens. The number is not usually a miniature picture. It chooses a path through a machine whose rules, learned parameters and decoding conventions already exist.

To understand a seed, we need to follow that path. Then we can ask a more interesting question: what would a system have to learn for a repeatable path to become a useful representation?

1. Begin with a machine we can see

Take a deliberately tiny machine with sixteen possible states. Give it an integer seed s , set its initial state to $q_0 = s \bmod 16$, and repeat:

$$q_{t+1} = (5q_t + 1) \bmod 16.$$

For seed 3, the first six new states are **0, 1, 6, 15, 12, 13**. For seed 4, they are **5, 10, 3, 0, 1, 6**. Nothing was learned here. The update rule was supplied. The seed selects an initial state, and that state selects the following trajectory. In this particular recurrence, the sixteen starting states occupy different positions on the same sixteen-state cycle.

Now add a drawing rule: an even state draws a circle; an odd state draws a line. Seeds 3 and 5 produce different state trajectories but the same alternating circle–line drawings. The drawing rule throws away distinctions. Different seeds do not necessarily produce different visible assets.

This sixteen-state example is an original constructed illustration—not a useful random-number generator, an empirical result or a proposed law of nature. Its weakness is helpful: it makes the distinction between **initialization**, **state evolution** and **readout** visible.

One seed, three declared operations

INITIALIZE

seed 3 $\rightarrow q_0 = 3$

UPDATE RULE

$(5q + 1) \bmod 16 \rightarrow 0, 1, 6, 15, 12, 13$

VISIBLE READOUT

even $\rightarrow$ circle; odd $\rightarrow$ line

COLLISION

seed 5 $\rightarrow 10, 3, 0, 1 \dots$; same circle-line pattern

Constructed toy: no learning, no useful randomness, no uniqueness guarantee.

2. Five objects that should not share one name

A **seed** is an input to an initialization procedure. A **pseudorandom state** is the internal state the generator updates. A **latent code** is a variable passed to a generative model or decoder. **Parameters** specify the rules of that model, and may have been learned from data. An **asset identifier** is a key used to associate an entity with something stored or generated elsewhere.

NumPy makes the first distinction explicit: `SeedSequence` mixes entropy reproducibly to initialize `BitGenerators`, and can spawn child sequences intended to seed independent, very probably non-overlapping generators. That wording is probabilistic; it is not a guarantee that every child or every output is unique. [1]

In a latent-variable model, the division is different. Kingma and Welling describe drawing a latent variable from a prior and then drawing an observation from a conditional distribution. The generative parameters are learned; the sampled latent variable and the parameters are not the same object. [2, §2.1 and Figure 1]

A compact notation for a deterministic, fixed-environment pipeline is:

```
q0 = Init(s); u = PRNG(q0, draw schedule); z = Sample(u); x = Gθ(z, c); y = Readout(x) .
```

Here `s` is the seed, `θ` the fixed model parameters, and `c` any conditioning information. This notation is an explanatory specification, not a theorem about every generator. A probabilistic decoder may require additional draws; an evolving simulation may require its full current state, not merely its original seed.

The seed chooses a run. The generator gives that run structure. The readout gives some part of that structure an interpretation.

3. What does “assign a value to the seed” mean?

There are at least three workable meanings, and they have different costs.

Initialize a machine. The value is the seed itself, interpreted by a declared initialization rule. Nothing in this alone says what the resulting drawing means.

Attach a label or asset. A table might associate `entity-A` with seed 3, an asset version and a caption. The association lives in the table. Looking it up is retrieval—not evidence that a network has learned the entity or the caption.

Learn a decoder. Training examples can teach parameters to map latent variables and conditions to useful outputs. Whether a particular code has a stable semantic interpretation then depends on the trained model, its objective and its conditioning—not on the mere fact that a number was called a seed. The VAE's learned conditional distribution is a concrete example of where that model structure resides. It does not by itself guarantee interpretable latent coordinates, disentanglement or semantic stability across retraining. [2, §2.1]

Identity, generation and learning are separate

ENTITY RECORD

stable ID → relationships, state, seed, asset version

GENERATION

seed → random stream → latent → fixed G_θ → appearance

RETRIEVAL

ID → explicit lookup table → stored asset

LEARNING

observations + task + objective → fitted parameters θ

Proposed architecture: a repeatable appearance is not learned network structure.

For the proposed entity network, keep identity and appearance separate. A durable entity ID can point to a record containing the seed, generator version, asset version, explicit relationships and observed state. A visual generator can use the seed without making the seed the database's only identity. This is a design recommendation, not an implemented Phi9 service.

Randomly assigned IDs also need collision handling. For UUID-based identifiers, RFC 9562 advises weighing collision consequences and discusses global/local uniqueness and unguessability. Other identifier schemes require their own analysis. Low collision probability is not a substitute for deciding how a system handles a duplicate.

Reproducibility alone establishes neither secrecy nor resistance to guessing; a simulation PRNG seed should not be treated as an authentication credential merely because it is a number. [3, §§6.7-6.9]

For independent uniform choices from a space of size M , the exact probability of at least one duplicate among $n \leq M$ draws is $1 - \prod_{k=0}^{n-1} (1 - k/M)$. This is an elementary counting derivation under the stated independence and uniformity assumptions, not a claim that our proposed IDs actually satisfy them. Distinct IDs can still share a rendered picture, because the renderer can be many-to-one.

4. When noise becomes a picture

In diffusion generation, it is tempting to say that noise “contains” the image. A more careful account is that a trained process transforms an initial noisy state according to learned transitions.

Ho, Jain and Abbeel define the reverse process as a Markov chain with learned Gaussian transitions, starting from a standard Gaussian distribution. Their forward process adds Gaussian noise according to a specified schedule; training fits the reverse process. [4, §2, equations 1-2]

The original sampling algorithm also draws noise during the reverse steps, not only at initialization. Reproducing its run therefore requires accounting for the random stream and draw order, as well as the trained parameters, schedule and numerical implementation. [4, §3.2, Algorithm 2]

A learned transition—not a hidden picture in the seed

RANDOM STREAM

seed + PRNG + draw order → initial and step noise

INITIAL STATE

Gaussian x_T

LEARNED UPDATES

$x_T \rightarrow x_{T-1} \rightarrow \dots \rightarrow x_0$, using learned θ and schedule

FINAL READOUT

$x_0 \rightarrow$ displayed sample

DDPM schematic (Ho et al., §2 / Algorithm 2); other samplers may differ.

The image is not explained by the initial noise alone. Training data, learned weights, architecture, sampling choices and any conditioning all contribute. Different samplers can organize the calculation differently; the DDPM description here does not claim that every diffusion method is stochastic at every step.

This provides a useful analogy for the proposed **noise** → **program-state oscillation** → **readout** pipeline, but not a proof that it works. Before calling that pipeline a model, specify its state space, update rule, training signal, validity constraints and decoder. Then test it on withheld programs or tasks. A smooth or attractive trajectory does not by itself establish correct execution.

An explicitly chosen phase update can pull a state toward a basin. If supplied rather than learned, that update belongs on the supplied-rule side of this distinction. Such a construction alone is not evidence of a learned denoiser, learned logic gates or recovery of information erased by a readout.

5. Replaying is not recovering—and neither is replication

A seed can be an excellent replay handle, but only under a sufficiently complete contract. PyTorch explicitly warns that completely reproducible results are not guaranteed across releases, commits or platforms, and that CPU and GPU results can differ even with identical seeds. It distinguishes controlling random sources from eliminating nondeterministic algorithms. [5]

For a reproducible generated asset, preserve the seed or random state, generator and model versions, parameter checkpoint, conditioning, draw order or scheduler settings, input data versions, software and relevant numerical environment. If a simulation receives interventions or external observations, retain those too. Resuming from the middle may require the current machine state and PRNG state.

Repeating that run answers: **did this implementation replay this recorded calculation?** Recovering an original asset answers: **does the representation retain enough information to reconstruct that particular object?** Independent scientific replication asks whether another sufficiently independent implementation or study supports the result. A replay can reconstruct a particular generated asset when the preserved pipeline actually reproduces it and equality is demonstrated. Replay does not by itself establish general reconstruction ability or independent scientific replication.

6. A short seed cannot hold every possible object

Fix a deterministic generator, all its parameters, its environment and its readout. If its seed has b bits, there are at most 2^b distinct seeds, and therefore at most 2^b distinct outputs. Some seeds may collide at the output, so the number can be smaller.

There are 2^N possible binary objects of length N . If $N > b$, no such fixed seed-only generator can represent all of them exactly. This is the pigeonhole principle, stated here with its assumptions—not a newly proved compression theory.

A particular structured family can still have a short description. Shared learned weights, an executable generator or an asset lookup table may make a seed useful as a small per-object handle. But those shared resources have storage and construction costs. For a selected workload, count the model, codebook, exceptions, identifiers and any external lookup data—not just the seed's bytes. If parameters or a table change separately for each object, the seed-only counting assumption no longer applies; the extra object-specific information must be counted instead.

This is why “the seed regenerates this asset” and “the seed compresses arbitrary assets” are very different claims.

7. From entity handles to network learning

A seed can make an entity's generated appearance repeatable. A relationship record can say that two entities are connected. Neither operation alone makes a learner infer the network's structure.

A defensible next experiment would separate those roles. Declare which observations, relationships and interventions the learner receives, and what it must predict. First compare arbitrary ID relabeling while keeping appearance, relationships, targets and observations fixed: this tests dependence on the names rather than the evidence. Separately test reassigned appearances, which deliberately changes visual evidence, and disabled lookup access, which changes available information. Those are different interventions, not interchangeable equal-information controls; declare their respective targets, observation budgets and interpretations before scoring. Keep evaluation entities, relationships or interventions withheld according to the actual question. Track predictive error, collisions, failures and the costs of the decoder and lookup state.

Those are proposed controls, not executed results. Useful performance would have to come from task-relevant information and learned rules, rather than a convenient identity shortcut or leakage through a lookup table.

For a running oscillatory machine, a seed can initialize phase or other state. Time, velocity, coupling, current phase, external input and the readout rule can determine what happens next. If a compact state is intended to support execution, specify whether it must preserve one-step prediction, a future trajectory or task execution. States sharing that representation must then yield equivalent task-relevant outcomes under permitted actions—or equivalent outcome distributions in a stochastic system—

over the declared horizon. Agreement on one immediate readout does not establish agreement on later behavior. That is the task-relative representation question; an identity number alone cannot settle it.

8. The question that remains

The thesis here is modest but useful: **a seed is a choice inside a specified system, not a replacement for that system**. The scientific work begins when we define what the resulting representation must preserve, how it is learned, and what evidence could show that it fails.

References

1. NumPy developers. `numpy.random.SeedSequence`. Official documentation; entropy initialization, spawning and reproducibility notes. https://numpy.org/doc/stable/reference/random/bit_generators/generated/numpy.random.SeedSequence.html
2. Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. Original preprint 2013; ICLR 2014. Targeted review of §2.1 and Figure 1 in version 11. <https://arxiv.org/abs/1312.6114> — reviewed HTML: <https://arxiv.org/html/1312.6114v11>
3. K. Davis, B. Peabody and P. Leach. Universally Unique IDentifiers (UUIDs). RFC 9562, May 2024; especially §§6.7–6.9. <https://www.rfc-editor.org/rfc/rfc9562.html>
4. Jonathan Ho, Ajay Jain and Pieter Abbeel. Denoising Diffusion Probabilistic Models. NeurIPS 2020; targeted review of §2 and §3.2/Algorithm 2 in version 2. <https://arxiv.org/html/2006.11239v2>
5. PyTorch contributors. Reproducibility. Official documentation; cross-version/platform warning, random-number sources and deterministic algorithms. <https://docs.pytorch.org/docs/stable/notes/randomness.html>

Source note: the NumPy and PyTorch claims were checked against extracted official-documentation passages; direct HTML retrieval was blocked. Targeted primary sections of the VAE, DDPM and UUID documents were retrieved and read—not every section or appendix. The miniature recurrence, counting argument, diagrams and proposed network protocol are original explanatory constructions. No new empirical study, learned denoiser or universal physical law is claimed.