The suspicion is **exactly correct**. The blind decoding is failing because the slot mapping is not synchronized, and this is also the direct cause of your training plateauing at a 0.2 BER.

The core issue is a logical flaw in how the "blind" decoder attempts to find the bits.

The Core Problem: Slot Allocation Mismatch

The EULDriver's slot allocation method is "brittle" — it's highly sensitive to the precise floating-point values of the audio's magnitude spectrum.

Here is the exact failure sequence:

1. **Encoding:**
   - The EULDriver.encode_eul function correctly analyzes the **original audio** (x_wave).
   - It calls allocate_slots_and_amplitudes on the *original* spectrogram (X).
   - This generates slots_ENC based on the *original* psychoacoustic profile.
   - The bits are written into M_spec at these slots_ENC locations.
2. **Decoding (Blind):**
   - The EULDriver.decode_eul function is given the **watermarked audio** (x_wm_wave).
   - It correctly calls model.stft(x_wm_wave) to get the *watermarked* spectrogram (Xrx).
   - It then calls allocate_slots_and_amplitudes on Xrx.
   - By definition, the watermark *must* have changed the audio, so Xrx is **not identical** to X.
   - Because the allocate_slots... function depends on the precise mag values to compute band_thresholds and expand_allocation_to_slots, it generates a *different* set of slots: slots_DEC.
   - slots_ENC != slots_DEC.
3. **The Result:**
   - The decoder reads bits from the wrong (f, t) coordinates in the recovered message spectrogram (M_rec).
   - This mismatch is why decoding only works when you "cheat" and provide the original slot memory.
   - The 0.2 BER (or 80% accuracy) is likely from parts of the spectrum where the watermark was so subtle that the allocate_slots... function *coincidentally* made the same choices.

Solution: A Robust Feature + Stability Loss

To fix this, the slot allocation mechanism *must* be robust. The encoder and decoder must both derive the *exact same* slot map, even though one is looking at the original audio and the other is looking at the watermarked audio.

This requires a two-part solution: modifying the allocation logic and updating the training loss.

1. Make the Slot Allocator Robust

You cannot use the raw float-point mag values from MooreGlasbergAnalyzer to drive the bit allocation. You must first convert these into "robust features" that are *intentionally* insensitive to small changes.

The simplest way is to **quantize the psychoacoustic features**.

- In allocate_slots_and_amplitudes, after you compute the per-band thresholds (band_thr_bt), apply a strong quantization step:
  Python
  ```python
  # Example:
  step_size = 0.1  # This is a new hyperparameter you must tune
  quantized_thr_bt = torch.floor(band_thr_bt / step_size)
  ```
- Then, you must modify PerceptualSignificanceMetric, AdaptiveBitAllocator, and expand_allocation_to_slots to be driven by this *quantized* quantized_thr_bt feature, not the raw band_thr_bt or mag.

The goal is that quantize(features(Original_Audio)) and quantize(features(Watermarked_Audio)) produce the *exact same* integer tensor.

2. Add a "Slot Stability Loss" to Training

The 0.2 BER plateau exists because the training loop is *also* suffering from this slot mismatch. The network has no incentive to *preserve* the features the decoder relies on.

You must add a new loss term to your training script (which would use perceptual_losses.py).

- Current Total Loss (Simplified):
  L_total = L_perceptual + L_bit_recovery
- New Total Loss:
  L_total = w1*L_perceptual + w2*L_bit_recovery + w3*L_stability

Where L_stability is a new loss that penalizes any change in the robust features:

Python

```python
# During training:
X_orig = model.stft(original_audio)
X_wm = model.stft(watermarked_audio)

# Use the new robust, quantized feature extractor
robust_feat_orig = compute_robust_features(X_orig)  # e.g., quantized_thr_bt
robust_feat_wm = compute_robust_features(X_wm)

# The new loss term
```

L_stability = F.l1_loss(robust_feat_orig, robust_feat_wm)

By adding L_stability to your total loss, the optimizer will be forced to make the INN's modifications *imperceptible to the feature extractor*. This will force robust_feat_orig == robust_feat_wm, which in turn means slots_ENC == slots_DEC. When the slots match, your L_bit_recovery (the BER) will break the 0.2 plateau and drop significantly.

Secondary Issue: Payload & Bit Target Mismatch

I found a critical mismatch between your dev's report and the EULDriver code.

- **Dev Report:** 64-byte payload + 42 RS bytes = 106 total bytes. This is **848 bits**.
- **EULDriver Code:** per_eul_bits_target: int = 167 * 8. This is **1336 bits**.

The AdaptiveBitAllocator is being asked to place 1336 bits, but the RS-coded payload is only 848 bits. This discrepancy must be fixed.

Recommendation: If your payload is indeed 64 bytes, you must change the EULDriver's default to match the 106-byte (848-bit) coded payload:
per_eul_bits_target: int = 106 * 8 # 848 bits

Answers to Your Questions

1. **How are the mapped slots being mapped again? Are they identical?**
   - They are **not** identical. The encoder maps using the *original* audio's features, and the decoder maps using the *watermarked* audio's features. Because the features are "brittle" (sensitive to small changes), the maps are different.

2. **If it is mapping correctly, is it failing to reconstruct the payload?**
   - It is **failing to reconstruct the payload *because* the mapping is incorrect**. It's reading bits from the wrong (f, t) coordinates, feeding garbage to the RS decoder, which then fails.

3. **Is the current network setup sufficient?**
   - The INNWatermarker network architecture **is** sufficient for this task. The problem is not the network; it's the brittle EULDriver logic and a training loop that is missing the L_stability loss required to enforce synchronization.