ChatGPT

Execution Plan for Project Resonance "Jericho"

1) One-Page Overview (Goal, Hypothesis, MVP, Timeline)

Goal: Develop an end-to-end **audio reasoning system** that takes raw audio (or STFT) as input and produces audio output, solving three algorithmic tasks **without any intermediate text representation**. The model must learn to perform discrete symbol logic entirely in the continuous audio domain for: - **Task 1:** Mirror (Copy/Repeat) – Repeat the input sequence (testing memory, including out-of-distribution new symbols and longer sequences). - **Task 2:** Bracket Legality (Logic Stack) – Determine if a sequence of brackets is balanced (testing implicit stack memory for nested structure). - **Task 3:** Arithmetic Mod (Composite Reasoning) – Compute arithmetic modulo operations, including multi-step compositions like `A % B % C` (testing multi-step sequential reasoning).

Key Hypothesis: A hybrid **Mini-JMamba** model (state-space layers + minimal attention) can learn algorithmic reasoning on continuous audio. By coupling an **audio representation encoder** (to reduce entropy) with a state-space model backbone (for long-range memory) and a constrained attention mechanism (for local content-based interactions), the system can internally emulate discrete logic (e.g. counting, stacking) without ever converting audio to text. We assume the model can generalize learned rules beyond the training distribution, which will be verified via **out-of-distribution (OOD) evaluations** (longer sequences, novel symbols, multi-step operations).

Minimal Viable Product (MVP): The MVP is a single-GPU prototype that **successfully learns Task 1 (Mirror)** end-to-end with audio. This includes a working data generator (audio sequences of symbols), the Mini-JMamba model producing an audio copy, and a scorer that decodes output audio back to symbols. The MVP should achieve near-perfect **in-distribution accuracy** on Task 1 (e.g. >95% exact match on test sequences of seen lengths/symbols) and show basic generalization (e.g. correctly copying a new symbol not seen in training, or handling a sequence slightly longer than trained on). This will prove the end-to-end approach and pipeline (data, model, training, scoring) are viable before scaling to Tasks 2 and 3.

Timeline (by week): - **Week 1: Data & Pipeline Setup** – Implement audio symbol generators for all tasks (start with Task1). Design the scorer to recover symbol sequences from audio. Verify the pipeline by generating a small dataset and having the scorer decode it. Train a trivial baseline (e.g. copy task with a small model) to ensure everything runs. MVP Checkpoint: simple model copies short sequences in audio. - **Week 2: Model Integration (Mini-JMamba)** – Implement the Mini-JMamba model: audio encoder/ downsampler, 10 SSM (Mamba-2) layers + 2 attention layers, and output decoder. Integrate the **auxiliary symbol loss** (e.g. CTC) alongside an audio reconstruction loss. Train on Task 1 with increasing sequence lengths. Target near 100% accuracy on in-distribution copy task. Evaluate on OOD length and new symbols to gauge generalization. Deliverable: Task 1 solved end-to-end; output audio clearly reproduces input sequence, verified by scorer. - **Week 3: Task 2 Introduction (Brackets)** – Extend data generator for bracket sequences (balanced vs. unbalanced). Train model (possibly multitask or separate run) on short bracket sequences to output a “valid” or “invalid” audio indicator. Introduce any required architecture support (e.g. a specialized output symbol or head for binary classification in audio form). Use curriculum: start with small depth. Evaluate bracket accuracy and analyze errors (especially false positives on tricky cases). Deliverable: Model can identify simple balanced vs. unbalanced sequences in audio (IID cases). - **Week 4: Task 3 Introduction (Mod Arithmetic)** – Develop generator for `A % B` (single-step mod) as audio sequences of digits and a "%" symbol. Train model (possibly add as another task in multitask training) to output the remainder in audio. Start with small numbers (1-2 digits) so model learns the basic operation. Evaluate correctness; ensure the model isn't just memorizing outcomes by testing on random values. Deliverable: Model correctly computes A mod B for simple cases (with audio inputs/outputs). - **Week 5: Scaling Up Complexity** – Increase difficulty on all tasks: - Task1: train on longer sequences (approaching the maximum we want in IID training) and more symbol variety. - Task2: longer bracket sequences (deeper nesting) and more challenging invalid cases. Possibly incorporate slight variations (ensure no trivial cues). - Task3: larger numbers (more digits) and introduce two-step compositions `A % B % C` in training. Monitor if model can perform sequential reasoning. Use a **curriculum scheduler** (Subject-Selector) to mix tasks and difficulty levels. By end of week, the model is trained on the full range of IID scenarios for each task. - **Week 6: OOD Evaluation & Tuning** – Freeze training and thoroughly evaluate on all **OOD splits**: - Longer sequences than seen in training (Task1, Task2, Task3 extended length or number size). - Novel symbols in Task1 (and any applicable novelty in others). - Multi-step compositions beyond training complexity (e.g. maybe 3-step mod if trained on 2-step). Diagnose failures: identify whether errors are due to reasoning (wrong symbol output) or rendering (audio issues). Conduct **negative control tests** (e.g. phase-scrambled inputs, label-shuffled training) to confirm the model isn't using shortcuts. Apply fixes or augmentations if needed (e.g. add phase randomization in training if it fails under that test, etc.). Deliverable: Report on generalization performance (e.g. >90% accuracy on 2x longer sequences, correct copying of new symbols, successful 2-step mod in ~90% cases). - **Week 7: Ablation Studies & Robustness** – Perform at least 5 ablations to validate each component's necessity: 1. Remove or replace attention layers (test pure SSM vs. hybrid). 2. Remove auxiliary symbol loss head. 3. Change the input representation (e.g. use raw waveform vs. spectrogram, or different downsampling factor). 4. Change the downsampling/encoder entropy reduction strategy (or remove it). 5. Alter curriculum (e.g. no curriculum or different mixing schedule). Possibly also: reduce state size or number of SSM layers to see performance impact, separate single-task models vs multitask model, etc. Analyze results to verify that e.g. attention helps OOD reasoning, auxiliary loss improves logic, etc. Deliverable: Ablation report identifying which design choices are critical for performance (for publication). - **Week 8: Final Integration & Documentation** – Consolidate the best-performing model and training configuration. Finalize the code repository with clear README, usage instructions, and scripts to reproduce training and evaluations. Prepare an **open-source release placeholder**: e.g. create a public repo with initial results (even if on a toy subset) and a timestamped claim of methodology. Write a brief report or blog demonstrating a sample input audio and the correct output audio for each task, along with metrics. Deliverable: First publishable result – e.g., a short demo of the model copying an unseen sequence, detecting bracket correctness, and computing a two-step mod, with accompanying metrics: **high IID accuracy (~99%)**, successful **OOD-length extrapolation (e.g. doubling sequence length with >90% success)**, correct **OOD-symbol copying (near 100% for truly novel symbols)**, and strong **compose-two-step success (perhaps ~85-95% exact accuracy)**.

2) Data (Task-wise Data Generation, Format, Curriculum, Splits)

We will design **synthetic data generators** for each task, ensuring we cover a wide range of cases and control for potential shortcuts. All data will ultimately be stored as pairs of **input waveform → output waveform**, along with the underlying symbol transcripts for evaluation. We choose a sample rate of **16 kHz** for raw audio generation (to cover audible frequencies) but will apply downsampling or framing in preprocessing as described later. Each task's data will include a clear definition of the symbol encoding in audio, the pattern to generate, and difficulty parameters (like sequence length, numeric size, composition steps). **Curriculum** is built into data generation: we generate simpler examples first and gradually increase complexity, marking data by difficulty level. We also define **IID vs OOD splits** explicitly to assess generalization.

Audio Symbol Encoding: To avoid using any text or ASR, we define a **mapping from discrete symbols to audio**. We will use simple synthetic tones for robust, unambiguous symbol representation. For example, assign each symbol (digit, letter, or special marker) a distinct tone frequency (e.g. in a musical scale or separate bands) so the model can reliably distinguish them by frequency content. Each symbol’ s audio could be a short sinusoidal beep (e.g. 100 ms duration) at a unique frequency. We will add slight variations in amplitude or phase per occurrence to prevent the model from memorizing exact waveform shapes. Specifically, we will **randomize phase** for each tone and add a bit of noise or jitter in duration to force the model to focus on spectral content (amplitude patterns) rather than specific waveform details (combatting overfit on waveform microstructure). The amplitude can be normalized such that all symbols have equal loudness, and short silences (e.g. 50 ms gaps) will separate symbols for clarity. The **scorer** (described later) will use these known frequencies to decode the sequence. This design ensures the audio is **compositional**: an input sequence’ s waveform is literally a concatenation of per-symbol waveforms, making the symbol sequence recoverable, but the model must learn the mapping and logic without any textual conversion.

Each task’ s data generator will create a large set of examples, saved as waveform files or in memory as numpy arrays, possibly alongside precomputed features (like spectrograms) if we choose to store those for speed. We’ ll maintain separate datasets for each task, or possibly a unified dataset with a task identifier if we do multitask training in one model. Given the tasks are algorithmic, we can generate practically unlimited data. We will generate enough training data to avoid overfitting (on the order of tens of thousands of examples per task at least), within storage and training time limits.

Below, we detail each task’ s data specifics:

- **Task 1: Mirror (Copy/Repeat)**
- **Input/Output Format:** The input is a sequence of symbols (e.g. letters or arbitrary tokens) represented in audio as described (distinct tone per symbol). The output should be the same sequence repeated or mirrored. We interpret “Copy+Repeat” as the fundamental operation of duplicating the sequence. For simplicity, we start with the output being **exactly the same sequence as input** (one-to-one copy). In a later curriculum stage, we can introduce a repeat-twice variation: output the input sequence twice in a row (testing a higher memory load). We will not use text at all; the audio itself is the input and output.
- **Data Generation:** We define a base alphabet of symbols for this task. For instance, use **10 distinct letters** (say A–J) for training. Each letter is assigned a tone (for example, A=440 Hz, B=480 Hz, etc., spaced out to avoid overlap). Generate random sequences of these letters of varying lengths. For training IID data, sequence lengths might range from e.g. 3 to 10 symbols initially (to keep audio length manageable). We ensure sequences are random combinations (uniform over symbols) so the model cannot cheat by, say, always expecting a particular letter next. For each sequence, the output target is the same sequence (for now). We also generate some variants if needed: e.g. another dataset where output is the sequence repeated twice (mirror-copy), to see if the model can generalize to that. This can be part of curriculum (first learn simple copy, then copy-twice).
- **Difficulty Curriculum:** Phase 1: short sequences (length 3-5) with symbols from a limited subset (maybe 5 letters) – easy memory load. Phase 2: increase length gradually (up to 10-15) and use all 10 training letters randomly. Phase 3: introduce “repeat output twice” if desired as a harder variant – the model must hold the entire input and replay it fully twice. Also possibly reverse-order output as an extreme case (mirror in the sense of reversal), but that might be beyond current spec; we will focus on straightforward copy to ensure clarity. We will also add **symbol variability**: The audio for each letter can have random phase each time and slight frequency modulation ($\pm$ a few Hz) or time-stretch to prevent rote waveform memorization. This forces the model to truly learn “which sequence of symbols” rather than memorizing the raw waveform.
- **Dataset Size:** We will generate a large pool, e.g. 50k examples, covering the different lengths and symbol combinations. Because the data is synthetic and inexpensive to generate, we can generate on-the-fly during training to avoid running out or overfitting specific sequences. Alternatively, we

pre-generate ~50k and shuffle. The **validation set** (IID) will have e.g. 1k examples drawn from the same distribution (randomly held out). The **test IID** similarly ~1k distinct examples.

- **Out-of-Distribution Splits:** For **OOD-Length**, we create test sequences longer than any seen in training. If training max length is 10, we might have OOD-length test at 15 and 20 symbols. We expect a good model to generalize the copy operation to these lengths if it truly learned the algorithm. For **OOD-Symbol**, we reserve a few symbols not present in training. For example, include letters K and L (with their own distinct frequencies) that never appear in training sequences. In an OOD-Symbol test, we give an input sequence that includes K or L (or is entirely made of new symbols) and check if the model simply echoes them correctly. A model that learned the concept “copy whatever symbol you hear” should succeed even on unseen symbols, whereas a model that memorized the training alphabet’ s patterns will fail. We will generate a small OOD-symbol test set (maybe 100-200 examples) containing unseen symbols in various positions (beginning, middle, etc.). We will also test **combined OOD** cases like longer sequences that also include a new symbol (this really stresses generalization).

- **Training/Validation/Testing:** The training set will contain only the base 10 symbols and lengths up to N (e.g. 10). Validation IID likewise. We’ ll maintain a separate **OOD evaluation set** not used in training: including OOD-length (with known symbols but lengths >10) and OOD-symbol (with new letters) as described. None of these OOD examples will be included in training or validation – they are purely for evaluating generalization.

- **Task 2: Bracket Legality (Logic Gate/Stack)**

- **Input/Output Format:** Input is a sequence of brackets `(` and `)` (we use a single type of bracket for the classic Dyck language problem of balanced parentheses). These will be encoded in audio as two distinct symbols, e.g. an “open bracket” tone vs a “close bracket” tone (for instance, `(` = 600 Hz beep, `)` = 800 Hz beep, ensuring they are easy to distinguish by frequency). The output is a very short audio indicating whether the input sequence is balanced (legal) or unbalanced (illegal). We choose to represent the output as a single symbol: e.g. use a special tone for “YES (valid)” and another for “NO (invalid)”. For example, a high-pitched ding for valid vs a low buzz for invalid, or simply assign two distinct frequencies (say 1000 Hz for "valid", 300 Hz for "invalid") so the scorer can decode a binary result. This means the model’ s output waveform for Task2 will be just one symbol long (plus maybe a fixed-length tone).
- **Data Generation:** We generate random bracket sequences and label them by correctness. Balanced sequences follow the Dyck-1 language rules (every prefix has #(`(`) ⩾ #(`)`), and total opens = total closes by end). Unbalanced sequences can fail either by a premature `)` or by unmatched extra `(` at the end. We will generate sequences systematically: for a given length L, generate some proportion of valid ones (which we can do via Catalan structures or brute force filter) and invalid ones (either by randomly inserting an error, or generating random sequences and filtering those not balanced). We ensure a mix of error types: sequences that fail immediately (first char is `)`), some that fail later (like `(())  )(` has a late error), and some that are nearly balanced except one extra bracket. We also ensure not to introduce any **spurious cues** – e.g. the distribution of lengths or symbols shouldn’ t allow trivial detection. Specifically, we will match the length distribution of valid and invalid examples so the model can’ t just learn “odd length means invalid” or similar. Also, half of the invalid cases will still have equal total number of `(` and `)` but in wrong order (to catch a model that might only count overall parity). This is crucial: a shortcut model could just count total opens vs closes; to prevent that, our invalid set includes many examples where counts match but order is wrong (e.g. `())(`). The model must learn to track the nesting properly (like a stack).
- **Difficulty Curriculum:** Phase 1: Start with short sequences (length 2 to 4) for which it’ s easy to manually balance or not. Many short sequences are trivial (like `()` is valid, `)(` is invalid). The

model can start learning basic pattern (maybe even a simple “illegal if starts with)” rule first). Phase 2: Increase length gradually (up to 6, 8, ...). With length, the complexity (nesting depth) can increase. We ensure to include edge cases like `((...))` perfectly nested, and tricky ones like `()()()` (valid with multiple components) or `()(())` etc., as well as typical invalid ones. Phase 3: By later training, reach sequences of length ~10 or more and maximum nesting depth ~5. This will push the model’s memory. We’ll also possibly incorporate logic gate subtask: The prompt mentioned “Logic Gate / Bracket” – to interpret this, we might include an easier sub-task in curriculum: for example, simple logic operations on bits. We could conceive a simpler audio task where input is like “1 AND 0” in audio and output is the result “0”. This could test a simpler form of logic without sequence memory. If we include it, it would involve encoding `0,1,AND,OR,NOT` as audio symbols and output a single bit audio. This might be an optional addition to help the model learn concept of logical operations. However, since the core is bracket checking, we may skip explicit logic gate tasks and focus on brackets (the mention might have been conceptual). We’ll primarily use bracket data; if needed, we can generate a small dataset of binary logic ops as a pre-training (to teach the network to output correct logical outcomes) but it might not be necessary.

- **Dataset Size:** Similar scale: we can generate, say, 50k bracket sequences for training. Balanced sequences are exponentially fewer than total sequences as length grows, but we can randomly sample until we have enough. We will approximately balance the dataset: 50% valid, 50% invalid, to avoid bias. Validation set ~1k examples, same distribution.
- **Out-of-Distribution Splits:** For brackets, **OOD-Length** is the key one: test on longer sequences (and deeper nesting) than seen in training. For example, if train max length = 10, we test on length 12 or 14. Balanced parentheses is a formal language that RNNs struggle with for much longer lengths, but we are interested in at least some extrapolation. Even a moderate increase will test if the model truly learned the stack rule or if it was pattern-matching within a narrow length. We don’t really have an **OOD-Symbol** here since only `(` and `)` are used. (We could theoretically test a different type of bracket symbol, like `[` and `]` instead of `(`, `)` to see if it generalizes the concept to a new symbol pair. This would be interesting: if we never trained on `[`/`]` but use the same audio frequencies as `(`/`)` just swapped, would it classify correctly? Probably trivial rename. Or use entirely new frequencies to see if it can apply the rule to novel tokens. We might include one experiment: train on one pair, test on a novel pair mapping. But it's not a primary goal since the logic is the same and symbol identity doesn’t matter if concept learned.)

- **IID/OOD Splits:** Training/validation contain lengths up to N (e.g. 10) with mix of valid/invalid. The IID test will similarly cover up to N. The OOD-length test will have sequences of length N+2, N+4 etc, balanced and unbalanced. We specifically will evaluate if the model’s **validity accuracy drops significantly as length goes beyond training**. Also, we will do targeted **diagnostics** on bracket task: e.g. test sets of cases where total count matches but order wrong (to ensure the model catches them). If the model were cheating by counting only, it would label those incorrectly as valid. So we will specifically measure accuracy on that subset.

- **Task 3: Arithmetic Mod (Composition)**

- **Input/Output Format:** Input consists of two or more integers and the modulo operator, output is the resulting integer of the modulo operation(s). We will represent numbers as sequences of digit symbols (0-9), and use a special symbol for the “%” operation. For example, input could be “**7 % 3**” (meaning 7 mod 3) or a two-step “**7 % 3 % 2**” (meaning first compute 7 mod 3 = 1, then 1 mod 2 = 1). In audio form, we might encode this sequence as: tone for “7”, tone for “(space) or operator”, tone for “%”, tone for “3”, etc. We need to clearly mark the separation between numbers and operations. A straightforward way: Use a dedicated symbol for the mod operator (e.g. a specific frequency beep for “%”). Possibly also use a short silence or a delimiter tone between separate operations for clarity (though the “%” symbol itself can serve as delimiter). Example encoding: “7” -> tone at 500 Hz (digit tones), “%” -> a distinct tone at say 1200 Hz, “3” -> tone at 440 Hz, etc. We’ll treat each digit 0-9 as a symbol with its own tone (maybe using a different frequency band than the letters of Task1 to avoid confusion if mixing tasks – but since they won’t appear in the same input, it’s fine to reuse concept as long as frequencies are distinct). The output is the result of the computation, which could be one or more digits. We will output it as the sequence of digit symbols corresponding to the remainder. For example, input “7 % 3” -> output “1” (one digit, with tone for “1”). If the result has multiple digits (e.g. 17 % 5 = 2, single digit; but if we did something that results in say 10, then two digits “1” “0”), we output each digit in order as separate symbol tones.

- **Data Generation:** For single-step modulo: randomly sample integers A and B within some range (e.g. 0 to 999 for A, 1 to 99 for B to avoid trivial B=0 or too large B). Compute R = A mod B. Ensure B is not zero (skip those). Represent A, B in decimal digits for input, with a "%" symbol between them. For example, if A=37, B=5, input audio sequence = [tone(3), tone(7), tone(%) , tone(5)]. Output sequence = [tone(2)] (since 37 mod 5 = 2). We generate a variety of combinations. To ensure model doesn’t just learn by rote, the range of values should be broad and not too small. Also, we can ensure that for any given B, there are multiple different A’s mod B to avoid a one-to-one mapping triviality. Ideally, the model should infer the mod calculation rule (which is essentially dividing and taking remainder). We will include cases where A < B (remainder = A itself), A = B (remainder 0), A just one more than B, etc., to cover edge cases. For multi-step (compositional) mod: generate triples (A, B, C). The interpretation is (A mod B) mod C. We compute it stepwise to get target result. We encode input as “A % B % C” (with two % symbols in between). For output, it’s again the final remainder as digits. We will generate such examples maybe only for training after the model learns single-step, or include them from the start if doing multitask. Possibly treat them as higher difficulty examples in curriculum.
- **Difficulty Curriculum:** Phase 1: small numbers with single-step. For example, A up to 20, B up to 9 (single-digit) – easy for model because these could even be done by lookup. But it establishes the concept and ensures output often single-digit too. Phase 2: increase A and B range (e.g. A up to 100, B up to 20). This introduces cases where A has two digits, B maybe two-digit, and output can be multi-digit (e.g. 100 mod 6 = 4, still one digit, but 50 mod 35 = 15, output two digits “1” “5”). The model must learn to output possibly two digits in sequence. We will also ensure to include those multi-digit outputs to practice. Phase 3: introduce two-step operations: A % B % C, initially with small values to keep it easier. The model has to conceptually do one mod then another. Many times, (A mod B) mod C = A mod B for independent random A, B, C? Actually not generally, so it’s a true two-step. Provide examples like A= (some), B, C to see if it can learn to effectively do first mod then mod the result by C. Phase 4: scale numbers further (A up to maybe 1000 or 9999, B up to e.g. 50-100, C up to similar). This results in longer audio input (A could be 4 digits, B 3 digits, etc.) which tests sequence handling, and more complex calculation needing multiple steps if we include composition. We won’t go too high due to audio length constraints and difficulty; just enough to test generalization (maybe up to 4-digit A in training, and test on perhaps 5-digit as OOD-length).
- **Dataset Size:** We will generate a wide variety, e.g. 50k examples that cover different ranges. We will likely stratify by difficulty: some fraction easy (single-digit, etc.), some medium, some multi-step, etc., to help curriculum. We might oversample multi-step examples as training progresses.
- **Out-of-Distribution Splits: OOD-Compose:** The primary OOD here is composition depth. For example, if we train mostly on 1-step and some 2-step operations, we can test a **3-step mod** (A % B % C % D) in evaluation to see if the model can generalize to an extra step it never saw. That would be a strong test of whether it learned a general algorithm for iterative mod or just specific patterns. Another OOD dimension is **number size** (which ties to length): if train max A is, say, 999 (3 digits), we test on a number like 12345 (5 digits) mod something. If the model truly treats each digit and does long division-like logic, it might handle a bigger number (though if it learned a

particular fixed memory, maybe not). So we can treat larger A (or B, C) than seen as OOD-length cases. Also **OOD-values**: for instance, maybe the combination of values in train is limited and we test a combination outside that distribution (less meaningful if we already randomly cover a lot).

- **IID/OOD Splits:** Training will cover up to certain digits and mostly 1-2 step. IID validation/test similarly. OOD test sets:
  - OOD-Length (numbers): e.g. if training had at most 3-digit numbers, test some 4 or 5 digit mod cases.
  - OOD-Compose: if training had at most 2-step, test a set of 3-step sequences like `W % X % Y % Z`.
  - Possibly OOD unseen digit? (Not really applicable; we use all 0-9 likely. We could exclude one digit from training to test copying a new digit in output. But that's not conceptually necessary because the model doesn't need concept of digit identity beyond symbol mapping – though for consistency with Task1 we might do something: maybe exclude digit "9" from training data, then in test give an example with 9 to see if it can handle a new digit. However, that might confound the arithmetic logic because if it never saw 9, it never had to compute something mod 9 or output 9. That might be too harsh and not a focus.)
- We ensure that training data has no direct leakage of output in input besides the legitimate information. For example, we won't feed in trivial cases like A % 1 too much (since that always yields 0, could be a trivial pattern, though including a few is fine to ensure model learns mod 1 = 0 rule). We also check distribution: ensure the most significant digit of A or such doesn't directly correlate with output in a trivial way beyond the real mod rule (shouldn't if data is uniform).

Data Storage & Format: We will store the datasets in a simple format, likely as .wav files or arrays plus metadata. However, to speed up training, we might not always load full waveform at 16kHz (which for a long sequence can be thousands of samples). Instead, we could store a **higher-level representation** (like a spectrogram or a downsampled version). But initially, for simplicity, we can generate raw wave on the fly each epoch (the cost is small for short synthetic audio). If memory allows, we can precompute features (like STFT magnitudes) for all training examples and store them in a NumPy array or HDF5 file so that training I/O is faster. Each example's input and output can be variable-length, so we will use padding or bucketing for batching. Likely we'll batch examples of similar length together to optimize (or use bucketing by length).

Negative Control Data: As part of our strategy, we may generate some **control datasets**: - For sanity, a set of random input–random output pairs (same formatting but output is incorrect) to train/test that the model doesn't learn anything if the mapping is random. This can confirm that any success is due to learning patterns, not data leakage. - Possibly a variant of Task1 data where input symbols and output symbols are deliberately shuffled (mismatched) to ensure the model only succeeds when mapping is correct. - Augmented data: apply **phase randomization** to some audio or inject noise to ensure model robustness (these augmentations can be part of training).

Dataset Summary & Estimates: We anticipate on the order of 50k training examples per task, which at an average input+output length of say 2 seconds (for moderate length sequences) at 16kHz is 32k samples per sec *2 = ~64k samples per example. 50k examples * 64k samples ≈ 3.2 billion samples total which is huge to store raw. But since sequences vary and many are shorter, and we can generate on the fly, storage is not a big issue. We'll likely generate on the fly or store in a compressed form (e.g. as symbol sequences that we can convert to audio on demand). Alternatively, store the symbol sequences and a script to synthesize audio on the fly in the training loop (this is efficient and ensures infinite variability with random phase).

Train/Val/Test Split: We will ensure strict separation: no exact sequence appears in train vs test (due to random gen that's easy). OOD sets are completely separate generation settings. A summary: - **Training**

set (IID): All tasks mixed or separate (depending on training strategy) with specified symbol sets and lengths. - **Validation set (IID):** For each task, 1k examples drawn from the same distribution as training (to tune hyperparams and check overfitting). - **Test IID:** Another 1-2k examples per task from same distribution for final reporting of in-distribution performance. - **Test OOD-length:** ~500 examples per task of longer sequences/numbers. - **Test OOD-symbol:** For Task1 (mirror) specifically, ~200 examples with unseen symbols as described. - **Test OOD-compose:** For Task3, ~200 examples of 3-step mod sequences (if training saw max 2-step). - **Additional diagnostics:** e.g. 100 crafted bracket cases to check specific failure modes (balanced count but invalid order).

All these splits will be saved or reproducibly generated with a fixed random seed to ensure consistency in evaluation.

3) Model (Mini-JMamba Architecture Design)

We propose a **Mini-JMamba** model tailored to these tasks, with a total depth of 12 layers, composed of **10 Mamba-2 (SSD) layers** and **2 attention layers**, as required. The design is inspired by the Mamba state-space architecture, which provides efficient long-sequence handling and has demonstrated strong performance on sequence tasks including audio modeling [1] [2]. Mamba-2 introduces **Selective State-Space Model (SSD) layers** that allow content-dependent gating of information [1], addressing a known weakness of pure SSMs in performing content-based reasoning [1]. We will leverage these SSM layers for their ability to propagate information linearly in O(T) time, combined with a small number of attention layers for focused content interaction. The attention will use a **sliding/chunked sparse pattern** to remain linear-time or O(T) in complexity, avoiding full quadratic attention on long audio sequences.

Overall Structure: It is essentially an **encoder-decoder** sequence transduction model, but implemented in a single unified architecture without explicit textual tokens. We treat the problem similar to sequence-to-sequence: the model must consume the entire input audio sequence and then produce an output audio sequence. We will handle this by a combination of encoding the input into a state, then decoding. Concretely, the model will function in two phases: 1. **Input Encoding:** Process the input audio frames through a stack of layers (mostly SSM, with possibly an attention layer here) to produce a latent representation capturing all necessary information (e.g. the sequence of symbols or the computed result). 2. **Output Generation (Decoding):** Generate the output audio from that latent representation. We implement this by either continuing the sequence processing with the same model (like an autoregressive generation using the state that now contains input info) or via a designated decoder block that attends to the encoder output.

Given the constraint of 12 total layers, we will allocate some for encoding and some for decoding: - We plan for **6 SSM layers in the encoder** and **4 SSM layers in the decoder**, which sums to 10 SSM layers. Then, insert the 2 attention layers strategically: one in the encoder and one in the decoder, or one as cross-attention between encoder and decoder and one as decoder self-attention. We opt for the latter to maximize utility: - **Attention Layer 1:** Encoder self-attention (sliding window) after a few SSM layers, to allow the encoder to capture any short-range content pattern that SSM might not catch quickly. Alternatively, it could be inserted as a cross-attention in the decoder. We will evaluate which is more beneficial. Since Mamba-2 layers already handle long dependencies fairly well (with input-dependent parameters), we might use attention more in the decoder. - **Attention Layer 2:** Decoder cross-attention to encoder outputs, if needed for the decoder to “see” the encoded input. However, we might not need an explicit cross-attn if we propagate state differently (see below). Another usage: decoder local self-attention to help it consider recently generated output frames for consistency (though SSM can also do that).

Given memory constraints and simplicity, an attractive approach is to not have a separate complex cross-attention, but instead **carry the hidden state from encoder to decoder via the SSM state**. Since SSM (state-space) layers can be run in a recurrent manner, we can feed the input sequence through SSM layers (with no output produced during encoding, just updating hidden states). At the end of input, the hidden state of each SSM layer will contain a summary (like how an RNN final state would). We then switch to output generation mode: feed a special start token or simply start the SSM recurrent loop again with no new input (or a dummy input) and have it generate outputs from its state. This approach means the same set of SSM layers can act as both encoder and decoder in time. However, implementing that seamlessly with our architecture requires careful control of state.

Alternatively, a clearer design: use the first 6 SSM layers (plus maybe an attention) as an **encoder** that processes the entire input sequence and outputs a sequence of hidden representations (like each time step’s encoding). Then, for decoding, use the remaining 4 SSM layers as a **decoder** that is run autoregressively: it will produce the output sequence one frame at a time, and at each step it can attend (via one cross-attention layer) to the encoder’s outputs to retrieve needed info (like keys/values from encoder sequence). This is analogous to Transformer encoder-decoder. The cross-attention can be chunked or restricted if input is long – e.g. we could allow the decoder to attend only to a summary or a reduced version of encoder output to keep it efficient. Another method: compress the encoder output via pooling such that we have a shorter memory for cross-attention.

We decide on a hybrid: We include a **Cross-Attention layer** in the decoder that uses a fixed sliding window over the encoder sequence. For instance, the decoder at output time step i will attend to encoder frames j that are within some window around relevant positions. But since the output might not align by positions (especially for arithmetic, the relevant part might be the whole input), we might instead use a simpler mechanism: use the final hidden state of the encoder as an initial state for the decoder SSM layers (this gives a fixed-size context vector, similar to older seq2seq). This is simple and avoids heavy attention, but might limit access to detailed input info. To mitigate that, we plan to incorporate at least one cross-attention: - Place **Attention Layer 1 as Cross-Attention** from decoder: it takes the output of encoder’s final hidden sequence as key/value and the current decoder hidden as query, and uses a local pattern or maybe full (since input length after downsampling might be moderate) to fetch relevant info. We will implement it as chunked attention if needed: e.g. split the encoder sequence into chunks or downsample it and allow global attention on that smaller set. Alternatively, use a learned compression of encoder outputs (like take one vector per input symbol by pooling frames).

- Place **Attention Layer 2 as Decoder Self-Attention** (within a window): to allow the decoder to consider its already generated output frames for consistency (though SSM can handle recurrence, a bit of self-attn might help ensure, for example, if output last two symbols need to relate, etc.). We restrict it to a small window (like the last few output frames) to avoid quadratic blow-up if output is long.

Encoder Downsampling & Entropy Reduction: At the very input, we include an **audio encoder** module to reduce the input sequence length and complexity. Processing raw 16kHz audio for several seconds is too long for even linear SSM layers and attention. So we will either convert to time-frequency representation or downsample in time: - Option 1: Use an STFT-based front-end. For example, compute a spectrogram with 10 ms frame hop (100 Hz frame rate), which compresses 16k samples to 100 frames per second (160x reduction). We might use 20ms windows for STFT to capture spectral detail. If we use magnitude spectrogram, we drop phase (randomize phase as augmentation). This yields e.g. a 257-bin magnitude vector per frame (if using 512-pt FFT). That is a lot of dimensions per time step. - Option 2: A learnable convolutional encoder: e.g. a 1D Conv with stride 4 or 8 to downsample raw waveform directly, possibly followed by another conv to reduce channels. For instance, a 1D Conv with kernel size 8, stride 4, 16 channels will reduce sequence length by 4x and produce 16-dim features per step. We can cascade two such layers to get 16x or 32x downsampling. We have to be mindful: too aggressive downsampling might lose timing needed for symbol boundaries. But since each symbol beep is ~100 ms, downsampling by 10x (to 10 ms frames) is acceptable. - We combine approaches: we will likely use a **learnable CNN encoder** that outputs a lower-frame-rate representation focusing on **amplitude envelope**. For example, first layer: Conv1D stride 4 (reduces 16kHz to 4kHz rep), then a second Conv1D stride 4 (down to 1kHz). This 1kHz (~1ms resolution) might still be high, we can go further or we can choose stride 8 once to get 2k. We' ll experiment with these hyperparams. The conv can also increase feature dimensionality (channels) to capture different frequency bands. Essentially, it can learn something akin to a filterbank. We could also initialize it to mimic an STFT filterbank (e.g. a fixed set of bandpass filters) if needed, but we can allow it to learn. - Additionally, we might explicitly emphasize **magnitude spectrum**: one trick could be to provide the model with a computed spectral magnitude as an input channel, discarding phase. For instance, compute a low-resolution FFT (maybe 64-point FFT every 10ms) just to get a coarse spectrum, and feed that as input features. This could guide the network to focus on spectral content (symbol identity) not waveform phase. However, doing this on the fly adds complexity. Simpler: incorporate an amplitude envelope extractor – e.g. we can square and low-pass the signal to get energy in the symbol's frequency band. - For MVP, we'll likely start with a straightforward approach: **mel-spectrogram front-end**. Compute, say, 80 mel-frequency bins every 10ms. So input = sequence of 80-dim vectors at 100 Hz rate. Then use a small linear or conv layer to reduce 80 dims to a smaller hidden size (like 16 or 32) that feeds into the Mamba layers. This drastically reduces sequence length (e.g. a 1-second audio is 100 frames) and also reduces data variation (phase invariance because using magnitude). This satisfies the requirement to **reduce entropy on input side** [3] [4] by focusing on magnitude spectrum. It is a manual feature but we might allow it since it's within policy (they said "or a learnable encoder do downsample"). - We will apply layer normalization or batch normalization on these features to ensure stable distribution into the model (especially since audio amplitudes could vary). - Summarizing input encoder: e.g. `Audio --> Spectrogram (or Conv) --> (optional linear) --> sequence length T' (much smaller than raw), feature dim D`. For example, raw 16000-length becomes T' ~ 100 for 1s audio, D ~ 32 features. T' could still be a few hundred for longer audio (like 5s -> 500 frames), which is fine for our model.

State-Space Backbone (Mamba-2 layers): We incorporate **Mamba-2 SSD layers** as our primary building blocks. Each such layer is essentially a Selective State-Space Model with learned parameters that can vary per time-step based on input [5] [6]. In practice, an SSD layer will take the sequence and produce a sequence of the same length, similar to a recurrent layer but computed efficiently. They have a hidden state dimension N (state size) which we can choose (Mamba-1 used N=16, Mamba-2 allows larger like 64 or 128 due to its efficient algorithm [7] [8]). We have to pick N to fit in our model size budget. Since our model is small (12 layers), we might use a moderately large N per layer to increase capacity for memorization (the blog noted that larger state size significantly improved tasks like associative recall [2]). Perhaps N=64 for each SSM layer is a target (we can adjust if memory too high). We also need to define the feature size (D) that flows between layers. Possibly, the SSM layers incorporate an implicit dimension (like the state dimension N is internal, while input/output could be lower dimension). The original Mamba architecture had no MLP or attention, but our hybrid does. We'll likely maintain a consistent vector size throughout layers (like model dimension d_model = maybe 128 or 256). The SSM state dimension N can be separate or equal to d_model; in many SSM implementations, the layer maps input (d_model) to N, runs the state update, then maps back to d_model. We can set d_model = maybe 128 for a start. - Each SSM layer will be followed by layer normalization (if needed) and maybe dropout (though with synthetic tasks, overfitting not huge, but could use small dropout to be safe). - We also include skip connections (like residual connections) around layers, as in modern architectures. The Mamba paper suggests an end-to-end architecture without MLP or attention can work [9], but we are adding minimal attention so it's fine. We just ensure the residual scaling is balanced.

Attention Layers: For attention, we need to implement **sliding or chunked attention**. Options: - **Sliding window self-attention:** e.g. each position attends to +/- w neighbors (like w=5 or 10 frames) rather than entire sequence. This gives local context linking. In audio, local correlation might help identify symbol transitions or boundaries. We can use this in the encoder around low-level layers. - **Chunked cross-attention:** for decoder, we can chunk the encoder memory. For example, if input is 500 frames, chunk into 5 chunks of 100 and allow decoder to attend one chunk at a time depending on progress, but since output doesn’t align easily, chunking cross-attn might be tricky. Alternatively, attend to a downsampled version: e.g. take every 10th encoder frame (or a learned average per 10) to get 50 length, and do full attention on that which is cheaper (50^2=2500, negligible). This provides a quasi-global view at low resolution. - Implementation wise, given time, we might use an existing efficient attention library or implement a mask for local attention in PyTorch. Memory 8GB can handle moderate lengths so we might not need to extremely optimize for MVP; a sliding window of say 50 frames on 500-length seq is 500*50 = 25k comparative operations, trivial. So sliding attn is fine. - Each attention layer will be multi-head (say 4 or 8 heads) with head dimension such that total equals model dim (e.g. 8 heads * 16 = 128). We will not use extremely large heads due to small model.

Decoder and Output Projection: After processing through decoder SSM and attention layers, we need to output audio. If we were doing direct waveform output, typically one would use upsampling or deconvolution to generate high-frequency details (like a vocoder). However, since we are prioritizing “symbolically correct” audio over high fidelity, we can simplify: - We can have the model output at the same representation level as input encoding (e.g. at 10ms frame rate with certain features), then synthesize a waveform from that. One way: have the model output an approximation of the spectrogram or even directly output the same representation as input had and invert it to waveform. But inversion of mel-spectrogram is not trivial without a Griffin-Lim or neural vocoder. Alternatively: - We constrain the output to the same discrete symbol audio style we used in data. That is, ideally the model will learn to produce similar tone waveforms for the symbols. We could design the decoder to actually produce parameters of those tones (like classification of symbol at each output position rather than raw samples). But that would be cheating to directly classify the symbol sequence (that’s basically solving it symbolically). We want it to produce an audio waveform that contains those tones.

One approach: **Frame-wise output** – Have the model output a sequence of log-magnitude spectra frames that correspond to the desired audio, and then reconstruct waveform with an external process. But to keep everything integrated and end-to-end differentiable, perhaps we have the model output at the same frame rate, then use an inverse short-time Fourier transform (ISTFT) operation (which is differentiable) to produce waveform. We can embed ISTFT as part of the model graph (PyTorch has some support for inverse stft). If our model outputs complex STFT bins or magnitude+phase, that’s heavy. Simpler: let model output a low-frequency representation and then we define a fixed synthesizer for the symbol tones: - For example, if each symbol corresponds to a known sine wave at a frequency, we could have the model output an indication of which frequency is active at each time step and then generate sine waves. But that’s basically forcing discrete output via continuous means. - Another tactic: since we know the target audio is a sequence of predefined tones (one per symbol), we can re-frame it as a **classification per output time step** problem: at each output position (which corresponds to a symbol position), the model should output the tone for that symbol. This suggests a simpler decoder: predict a sequence of symbol tokens (like classification) and then separately generate the tone audio for those tokens. However, that breaks end-to-end audio training unless we integrate it differently.

To maintain end-to-end training, we will do the following: **Dual-output heads** – The model will have: - a **Waveform output head** that tries to directly produce the waveform (or a filtered version). Perhaps a linear projection from model dimension to a waveform sample or frame, and then use a convolution to upsample frames to waveform. But generating waveform from such a small model is challenging (like a neural vocoder). - Instead of raw waveform, we can have it output a **low-frequency approximation** that is enough for the scorer to decode symbols. For instance, the model could output a downsampled waveform at 1kHz that mainly contains the envelope or fundamental frequency content. The scorer could still decode symbol presence from that (since we choose distinct frequencies well below Nyquist of 500 Hz if downsampled to 1kHz, though if symbol tone is at 600 Hz that's above 500, not good). Alternatively, model outputs a spectrogram which includes those tone frequencies.

Given complexity, the pragmatic approach: **train the model to output at the same feature level as input (mel spectrogram)**. Then separately, we use a simple synthesizer to convert that to audio. But then the training loss has to compare generated spectrogram to target spectrogram, not waveform directly. That’s acceptable (multi-resolution STFT loss is essentially comparing spectrograms). We can do: model outputs a log-magnitude spectrogram (e.g. 80 mel bins every 10ms) for the output audio. In training, we compute the spectrogram of the target output audio and apply an L1 or L2 loss on that (this is a form of spectral reconstruction loss). Additionally, perhaps use a smaller L1 on waveform as auxiliary to ensure correct phase alignment of tones.

However, implementing ISTFT for perfect waveform might be optional if our metrics (exact match etc.) rely on symbol content rather than waveform alignment. The scorer does not necessarily need the exact waveform as long as it can decode the symbol sequence. If the model’s output spectrogram clearly has peaks at the symbol tone frequencies in correct order, the scorer will pick it up. This means high-fidelity waveform isn't required, as stated. So the model could output somewhat “buzzy” or simplified audio and still get full score if the symbol frequencies and timing are correct.

Therefore, we will design the output path as follows: The final decoder layer(s) produce a sequence of vectors (one per output time frame, e.g. per 10ms). These go through a linear projection to output a smaller feature, e.g. n_mel (80) values representing log-mel spectrogram at that frame. We can then either: - Immediately compare to target mel spectrogram (if we have target mel). - Or apply an inverse mel filter and overlap-add to reconstruct approximate waveform to compare to target waveform in time domain for L1 loss.

We might implement both losses (time and freq) to guide properly.

Model Diagram Summary (textual):

```
Input audio (waveform 16k)
 -> [Preprocessing: spectrogram or conv encoder] -> sequence length T_in' (~100-500), feature
dim ~32
 -> [Encoder: 6 layers of SSM (Mamba-2) + 1 local attention layer] -> encoded representation
(length T_in', dim d_model)
 -> [Option A: final encoder hidden state aggregated as context]
 -> [Decoder: uses encoder output]
    If using cross-attn:
       - [Cross-Attn layer: Query=decoder state, Key=Value=encoder outputs (with maybe
downsampling or window)]
    -> [4 layers of SSM + maybe 1 decoder self-attn] -> decoder hidden sequence (length
T_out', dim d_model)
 -> [Output Head: linear projection] -> predicted output spectrogram frames (length T_out')
 -> [ISTFT or fixed decoder] -> predicted waveform output (optional during training)
```

We also have an **auxiliary symbol classification head** (see next section) attached either at the output or intermediate.

Mamba-2 (SSD) Layer implementation: We will likely use an existing implementation (maybe from the GitHub state-spaces/mamba [10]) to avoid coding the SSM from scratch given complexity. The layer will take input (batch, length, dim) and have internal state N. We must ensure it can run on GPU efficiently (maybe uses CUDA kernels as per authors). If issues, we might fallback to a simpler S4 or even an LSTM for MVP to ensure progress, but the final plan is to use Mamba-2.

Parameters & Memory: With d_model ~128, state N=64, 12 layers, plus attention heads, the model might be around a few million parameters, which is fine for 8GB GPU. The memory usage is mostly from the sequence lengths; we have that controlled by downsampling. We will also use mixed precision to cut memory (details in training section).

Adaptability: If this architecture struggles, alternatives include: - Reducing reliance on cross-attn by making the SSM layers process input and output in one pass (like prefix LM style). - If one model multi-task is too hard, we might train separate models per task as ablation. But prefer single model.

Auxiliary components: We include a **scorer module** outside the model for evaluation (not affecting model forward). That is separate, described in losses. Also, we might include any needed task ID encoding if tasks differ: e.g. possibly prepend a special token or feature to indicate which task (so model can switch behavior). But since tasks have disjoint symbol sets (letters vs brackets vs digits), the model can likely infer task type from input content. We will not explicitly use a task token to avoid leaking that info (though it’ s not a leak, it could help multi-task learning to have an indicator). If we see confusion, we might add a one-hot task embedding appended to encoder input.

Inserting Attention Layers: Ensuring they are not full O(T^2): - Encoder attention: we set a window size w (maybe 5 or 10 frames) for local self-attention. So each position attends to w frames before and after. This is implemented with a mask in attention or a convolution-like attention. - Decoder cross-attention: if we allow full attention over encoder output, that’ s O(T_enc * T_dec). If T_enc ~ 200 and T_dec ~ 50 (for moderate lengths), that’ s 10k, which is actually not large at all. Even 500 x 100 = 50k, trivial. So we might get away with full cross-attn without chunking for our scale. But to abide by spec strictly, we can say we chunk encoder sequence by, say, splitting it into 5 chunks and attending each chunk separately (not typical). Alternatively, restrict each output to attend only to encoder frames within a certain range around some anchor. Hard to define anchor because output index doesn’ t directly align with input index in tasks like mod (though in copy task, output index = input index if copying one-to-one). - Actually, for copy task, cross-attn isn't even needed if we just carry state; for mod, the entire input influences output (like both numbers). - Possibly simpler: let decoder cross-attend to the **final encoder state (vector)** only, i.e. not using a full sequence but one context vector. But one vector might not hold full sequence details for tasks like copying entire sequence. So not viable for Mirror. - So we do need some form of full encoder sequence access for copy task specifically, because the model needs to output each symbol it saw. That basically requires a memory of the whole input sequence. Mamba-2 layers might have effectively stored the input in their state though. If we treat the entire model as one big recurrent machine, it could just replay from memory. But in practice, easier if the decoder can explicitly attend back. - Therefore, for Task1 within the model: we might do a trick – since copy output aligns to input, we could just shift the input through and let it output at same time (like an echo RNN). But to keep uniform approach, we'll rely on the model learning to store input in state and output via cross-attn retrieval. - So the decoder cross-attn should be allowed to attend broadly. We can allow **global attention but with downsampled keys**: e.g. use every 2nd or 4th encoder frame as keys (so 200 frames -> 100 or 50 keys). This still lets it see the sequence at coarse level, which might or might not suffice to output every symbol accurately. Alternatively, we do local but the decoder moves a window along input as it generates output sequentially. - Perhaps for copying, since output length = input length, we can align them by position in a straightforward way using the state approach (like RNN memory). If the SSM layers are truly like a big RNN, the simplest is: run the SSM in a single pass where it outputs the symbol it just saw one step later (like an echo). That would handle copying easily but wouldn’t handle mod or bracket which need to wait until end to decide output. So we might incorporate a mechanism: maybe train the model such that for Task1 it learns to copy, for others it waits. - We can incorporate an **attention in decoder that at output step i attends to encoder step i (for copy)** – that is effectively identity mapping. But we need something more general for tasks where output shorter. - Possibly use different decoding strategies per task: e.g. for copy, treat it as concurrent mapping, for bracket and mod, treat it as needing full input then output one symbol (like sequence to one). The model might handle that implicitly if its decoder can produce variable lengths.

Summarizing: We might refine design after initial experiments, but the core is: - 12 layers: 10 SSM (powerful for sequence memory) + 2 lightweight attentions (for content-based linking). - Input embed/ downsample to reduce length drastically (target ~100 frames/sec or less). - Output predicted as sequence of spectral frames, not necessarily raw waveform. - A small feed-forward head to map hidden to output features. - Possibly an **auxiliary symbol prediction head** branching off to help training (discussed next).

4) Loss Functions (Main Loss + Auxiliary Reasoning Loss)

Our training objective will combine a **primary reconstruction loss** on the output audio with an **auxiliary loss that directly guides the model’s reasoning at the symbol level**. We need to carefully design these to ensure we improve learning without “leaking” the answers in a trivial way.

Primary Loss (Audio Reconstruction): This measures how close the model’s output audio is to the target audio. We will use a combination of time-domain and frequency-domain losses to capture both overall waveform shape and the spectral content: - **Waveform L1 Loss:** Compute the L1 error between the predicted waveform and the target waveform (after appropriate alignment). Given our output might be frame-based, we might reconstruct a waveform first. Alternatively, compute L1 on a downsampled waveform or directly on the predicted spectrogram vs target spectrogram in linear domain. L1 in time encourages correct amplitude and phase alignment for transients. - **Multi-Resolution STFT Loss:** This is a common choice in audio generation tasks [11]. We will calculate the STFT of both predicted and target audio at multiple resolutions (different FFT sizes and hop lengths) and compute spectral convergence and log magnitude losses [11]. Specifically, define spectrograms $S(f)$ for a few settings (e.g. FFT 512 hop 120, FFT 256 hop 60, etc.). The STFT loss includes: - Spectral convergence: $| S_{pred} - S_{target} |F / | S | F$, - Log-magnitude loss: L1 between $\log|S|$. We sum these for a set of resolutions [11]. This loss ensures the model gets the frequency content of each symbol right (peak at specific freq) and the overall envelope timing right, even if the phase is off (phase differences affect time waveform L1 but less so log mag). - }|$ and $\log|S_{target**Mel Spectrogram L2 Loss (optional):** In addition, since we explicitly may predict mel-spectrogram, we can directly do an L2 or L1 on the mel spectrogram frames output by the model versus those of target. This overlaps with multi-res STFT (since one resolution might effectively be a mel filter). - We will weight these losses to balance. The STFT loss often is weighted quite high in audio tasks because it correlates better with perceptual quality [11]. We might use e.g. 0.5 for waveform L1 and 1.0 for STFT loss sum, as a starting point.

These reconstruction losses will drive the model to produce an audio that sounds like the target, meaning it has the right tones at the right times. If the model gets the logic wrong (e.g. outputs wrong symbol tone), these losses will be high because frequencies won't match at those times (log-mag loss will punish that). Thus, the model is incentivized to get the correct symbol sequence and timing.

However, the reconstruction loss alone might be difficult for the model to navigate, because generating exact audio is tough. It might struggle to figure out the correct sequence purely through this error signal, which can be sparse (e.g. if one symbol is wrong frequency, loss gradients might not directly tell it "you chose wrong symbol", it just sees spectral difference). This is where the auxiliary loss helps.

Auxiliary Loss (Reasoning / Symbol-level): We introduce an auxiliary objective that explicitly trains the model to produce the correct symbol sequence, without bypassing the audio modality. Two possible approaches: - **CTC (Connectionist Temporal Classification) Loss on Symbol Sequence:** We can attach a classifier that reads the model' s **predicted audio output** (or an intermediate representation thereof) and outputs a sequence of symbols, and use CTC to align with the target symbol sequence. Concretely, we can take the final decoder hidden states (one per frame, e.g. 100 Hz frames) and add a small fully-connected layer to predict a probability distribution over the symbol vocabulary (including a blank symbol for CTC). The target sequence is the known symbol string (like "A B C" or "valid"/"invalid" as single token). We then use CTC loss which will force the model' s hidden states to have peaks corresponding to the target symbols in order. Essentially, this is training an ASR-like decoder on the model' s output. If the model output audio is correct, a good classifier could easily pick out symbols; by training with CTC, we encourage the model to make its internal representation (or output acoustic patterns) decodable as the correct sequence [11]. This provides a guide: the model doesn' t have to figure out the symbol boundaries alone, CTC nudges it to produce distinguishable symbols in order.

Why CTC and not simple cross-entropy on frames? Because the number of output frames and their alignment to symbols is not fixed. The model could stretch a symbol over multiple frames or merge them. CTC handles that by allowing flexible alignment. It uses the highest probability path matching the sequence.

We will be careful: We do **not feed the ground-truth symbols directly into the model at any point**; we only use them to compute this loss from the model' s own outputs. The classifier head for CTC looks at model' s features (which depend only on input audio), so it cannot magically introduce information the model didn' t compute. It just provides a clearer training signal: e.g. if the model' s output audio was supposed to have a 600 Hz tone for "(", but it's off, the CTC will see that the model' s hidden representation isn't matching symbol "(" and backpropagate to adjust it. This helps the model learn the mapping from input to correct output symbols more directly, rather than solely via spectral error which is indirect.

There's a subtlety: Could the model cheat by using this classifier to output symbols without actually producing them in audio? Potentially, if the classifier head is too "powerful," the model might minimize CTC by encoding symbolic info in a way the classifier can read, even if audio is off. To mitigate that, the classifier should use features closely tied to the generated audio. Ideally, we use the same features that go into generating audio. For instance, if the model outputs a spectrogram, we can apply a small CNN or even threshold to detect peaks at known symbol frequencies in that spectrogram and apply CTC on that. That ensures any symbol prediction correlates with actual spectral content. Alternatively, take the decoder hidden state just before output projection – but if the model learns to encode symbol identity in hidden state but fails to output it in waveform, we might satisfy CTC but not waveform loss. However, the waveform loss would still penalize a missing tone, so the model can' t fully ignore audio.

We will weigh CTC auxiliary loss relatively lower than main loss to ensure model doesn' t focus solely on pleasing CTC at expense of actual audio output. For example, weight it say 0.2 of main loss, tuning as needed. The aim is to use it as a hint, not the primary driver.

- **Segment-wise Cross-Entropy:** If we have alignment (which we do, since we know each output symbol' s approximate position in time for training), we could simplify by cutting the output into segments (one per symbol) and adding a classification loss for the symbol in each segment. For example, in Task1, each output symbol corresponds to a specific time interval; we could force the model’ s representation in that interval to classify as that symbol. But alignment might not be trivial for mod tasks where output length is shorter than input.

CTC is more flexible, so we prefer CTC. It's widely used to train models to produce sequences from audio and fits our end-to-end ethos.

Why this helps reasoning: The auxiliary symbol loss **does not provide the correct answer as an input**, it only evaluates the model’ s output (or late-stage representation) against the correct symbol sequence. It’ s akin to having a differentiable grader that tells the model more directly if each symbol is right or wrong, instead of only telling it via audio differences. This should accelerate learning of the correct discrete operations: - For example, in Task3 (mod), the model might initially guess wrong remainder; the audio loss will be a bit higher because frequencies differ, but CTC loss will explicitly be high until it outputs the correct digits. CTC gradient will push it toward outputting the correct digits' features. It still has to figure out the logic from input because nothing in this loss tells it how to get the right symbol, just whether the output is right. So it **does not leak the solution procedure**, only provides a clearer error signal for the output.

By designing it carefully, we avoid it bypassing the audio: the model can’ t just memorize input→symbol mapping because it still must produce audio that aligns with those symbols for the loss to decrease. And since we do not input text anywhere, it can’ t cheat by e.g. copying text from input to output. The symbol loss is purely on output.

Implementation detail: We will implement a linear layer on top of either the decoder hidden states or the predicted spectrogram. Possibly, we can take the predicted mel-spectrogram at each frame and for each of our known symbol frequencies (for tasks where output is multi-symbol, like copy and mod) detect energy. But a linear layer on hidden should suffice: it will learn to correlate hidden features with particular symbol identity. We'll include a "blank" symbol in CTC for frames that are between symbols (or for Task2 which has single output, the sequence is length 1 so CTC is trivial there anyway).

We will apply the auxiliary loss to tasks that have sequential outputs (Task1 and Task3 mainly). For Task2 (bracket validity), the output is one symbol. CTC isn’ t needed (no sequence, just one token). We could simply apply a cross-entropy on the final output classification (valid vs invalid). In fact, for Task2, we will do that: treat “valid” vs “invalid” as a 2-class classification and use cross-entropy loss in addition to audio loss. However, since the output also has an audio form (two different tones), the audio loss itself might suffice, but to help the model early on we can add a small CE loss on the final hidden state vs the true class. That is another auxiliary – essentially teaching it yes/no in a simpler way. We must be careful that this doesn’ t leak: but it’ s not leaking any new info; it's just supervising the model’ s internal state to have the correct class ready. It might allow the model to solve bracket via that head without making the audio correct. But the audio loss will ensure it still has to produce the tone. Actually, if the model learns bracket correctly, it could minimize CE by having correct internal output, and might slack on generating the actual tone if audio loss weight is low. So we will keep audio loss high to force it to actually emit the correct tone. Possibly weight this bracket CE loss low (like 0.1) just to guide early learning.

Summary of Losses: - L_main = L1_waveform * α + L_multires_STFT * β (with α, β weights, e.g. α=1, β=1). - L_aux1 = L_CTC(symbol_sequence) * γ (for tasks with sequence output, e.g. γ=0.5). - L_aux2 = Cross-entropy on final classification for Task2 * δ (maybe δ=0.5). - Possibly separate weighting per task if multi-task (we might sum losses from each task, but easier is to construct a unified loss where each example’ s relevant loss is counted). - The combination is optimized jointly. We’ ll tune weights so that at the beginning, the auxiliary provides strong signal to get the logic right, but as training progresses, the audio loss dominates to ensure fidelity.

Preventing answer leakage: The key is we **do not use the scorer or ground truth sequence in any way that bypasses audio generation.** The auxiliary head is essentially making the model do part of the scorer’s job internally – which is fine because it still has to output audio. We ensure the auxiliary head is only looking at model’s internal representation after it has processed input (so it must infer answer from input itself). - We will also monitor if the model might rely too much on the auxiliary: e.g. if we see it predicting correct symbols in the CTC head but audio is gibberish, that’s a sign of mismatch. Our design with joint loss should avoid that because gibberish audio will incur high main loss. - In worst case, we can force some coupling: e.g. feed the output spectrogram into the CTC classifier instead of hidden state, to ensure the model must actually produce the correct spectrogram for CTC to succeed. This is a possible improvement: run a small differentiable decoder (like a lightweight ASR) on the predicted audio to get symbols and do CE on that. But that’s essentially training a separate model. Simpler is as we have.

Additionally, we incorporate **penalty losses for cheating behavior** if needed: - For example, if we suspect the model tries to encode outputs in inaudible high-frequency (above what scorer listens to), we could add a loss to discourage energy outside symbol bands (like a high-frequency energy penalty). We will monitor spectrum; likely not needed if frequencies are fixed and we limit model output range. - If the model outputs nothing (silence) except the classifier head does everything (in a pathological scenario), the audio loss (especially STFT) will penalize silence vs expected tone hugely, so that won't be minimal, preventing that cheat.

In summary, our loss function encourages the model to **produce correct outputs and make them easily decodable**:

> “We combine a time-domain L1 and multi-resolution STFT loss to ensure the waveform output matches target audio, and an auxiliary CTC loss on symbol sequences to explicitly guide the model toward correct symbolic outputs without leaking ground truth answers into the model’s inputs or intermediate layers.”

This multi-task loss approach is intended to improve reasoning. The CTC essentially asks the model to **explain its output in terms of symbols** during training. This should help it avoid simply mimicking audio without understanding. It’s similar to forcing a neural TTS model to also output the transcript it’s speaking – it aligns acoustic generation with symbolic content, hopefully improving clarity and correctness of content, not just acoustic similarity.

5) Training Plan (Batching, Optimizer, Schedules, Curriculum Implementation)

Training this model will require careful management of sequence lengths and task mixing, as well as standard best practices (mixed precision, grad clipping, etc.). Here we outline the training configuration:

Batching and Sequence Length: We will use dynamic batching to handle varying sequence lengths. Likely we separate tasks by batch or ensure examples in a batch have similar lengths to minimize padding. We might maintain a smaller batch size for very long sequences to avoid memory overflow. - Initially, for MVP and early curriculum (short sequences), we can use a larger batch size (e.g. 32 or 64 examples of short length). - As sequence lengths and model complexity grow, memory might force us to reduce batch size (maybe down to 8 or 16 when sequences are long ~5s). - We will adjust batch size dynamically or per epoch (for example, while training on shorter tasks we can accumulate gradients to simulate larger batch if needed).

Given GPU 8GB, and model ~ millions of params, a rough guess: 16 examples of 1-second audio with spectrogram input (100 frames) might easily fit. For 5-second, 16 might still fit since 500 frames * 16 = 8000 sequence length total in attention, which is fine for linear layers and manageable for local attention. We should also consider using **gradient accumulation** if we want an effective larger batch but can't fit more at once. E.g. accumulate gradients over 4 mini-batches of 8 to simulate batch 32.

Mixed Precision: We will use **AMP (Automatic Mixed Precision)** to accelerate training and reduce memory. All matrix multiplies (in SSM, attention, etc.) can be float16 except perhaps the state updates if they risk precision issues (we will test stability; Mamba’ s recurrent operations might need float32 to avoid numeric issues, but we can keep those in FP32 and others in FP16). PyTorch AMP can handle that with appropriate contexts. Mixed precision should roughly halve memory usage and double speed on 4070 (which has Tensor Cores).

Optimizer: We will use a proven optimizer like **AdamW** (Adam with decoupled weight decay) for stable training. Adam is good for these models, and we can set initial learning rate based on model size and batch. Possibly start around LR = 1e-3 or 5e-4 for the smaller model. Since we have multiple losses, they’ re summed, but that doesn’ t change optim fundamentally. - We’ ll use weight decay (small, e.g. 1e-2 or 1e-3) to regularize a bit. - We may use gradient clipping (norm clip at perhaps 1.0 or 5.0) to prevent any exploding gradients, especially since SSMs can sometimes produce large gradients if unstable. This also protects against occasional large error outliers from e.g. a long sequence.

Learning Rate Schedule: Given the tasks aren’ t extremely large-scale, a simple schedule might suffice: - We can do a short **warmup** (like 1000 steps) where LR linearly increases from 0 to target (to avoid unstable early training). - Then use either **cosine decay** or a fixed LR with manual decay on plateaus. A cosine schedule (decay to very low over course of training) is popular for transformers and likely fine here. Alternatively, since we might curriculum and train for a fixed number of epochs, we could drop LR by factor 10 halfway, etc. - We'll monitor validation. If validation loss plateaus, we can reduce LR. - Estimated total training steps might be on the order of, say, 100k steps for full multi-task training to converge logic (just a rough guess). We’ ll adjust as we see training curves.

Training Curriculum & Task Mixing: This is crucial. We have three tasks, each with difficulties. We must ensure the model doesn’ t overfit one or ignore another. We propose a staged curriculum: - **Phase 1 (Weeks 1-2):** Train Task1 (Mirror) alone, starting with short sequences. This warms up the model’ s basic ability to copy memory and produce output audio. We might do this phase with just Task1 data for say a certain number of steps (e.g. 10k steps) until it achieves high accuracy on short sequences. This ensures the model and pipeline works. We keep the other tasks aside to not confuse early training. - **Phase 2:** Introduce Task2 (Bracket) either in a multi-task fashion or sequentially. We could either: - Continue training the same model but now with a mix of Task1 and Task2 batches. Or - Train Task2 separately to convergence, then combine. We prefer multi-task from the point we introduce second task, to start teaching the model to differentiate tasks. We will implement a **task sampler** (Subject-Selector) that decides whether a given training batch is Task1 or Task2. For example, use a proportion – initially, because Task1 is already partly learned, we might allocate say 70% of batches to Task2 (new) and 30% to Task1 (to maintain skill) during this phase. Within Task2 batches, we also curriculum the lengths as described (starting short). This approach is one interpretation of the “Subject-Selector”: an automated way to choose the next training sample’ s task based on some criteria. We may adjust those percentages based on validation (if Task1 performance starts dropping, increase its fraction). - **Phase 3:** Introduce Task3 (Mod) similarly. Now we have 3 tasks. We will likely allocate batch proportions like 50% mod, 25% mirror, 25% bracket initially since mod might be hardest/new. Over time, adjust if needed. We can also do round-robin: e.g. one batch Task1, next Task2, next Task3, etc., which ensures each iteration sees a different task. Alternatively, each epoch could cycle through tasks. We'll likely randomize per batch with fixed probabilities. - **Dynamic Difficulty Increase:** Within each task, we use curriculum. We can implement this by having our data generator or dataset capable of sampling by difficulty. For instance, we label examples or generate them on the fly given a current max difficulty parameter. We then gradually raise that parameter. Specifically, for each task: - Mirror: start with max length L=5. After a certain epoch or when training accuracy > X, increase to L=7, then 10, etc. We might schedule it by epoch: e.g. epoch1: L=5, epoch2: L=7, epoch3: L=10. Or use a performance trigger: if validation copy accuracy for length 5 is > 90%, then allow length 7 in training. - Bracket: similarly increase max length/nesting. - Mod: increase number size and whether 2-step included. E.g. train on single-step small numbers until performance high, then add some 2-step examples, etc.

We can implement a simple schedule: For first N epochs, use easy settings, next N use medium, etc. Or gradually linearly increase some difficulty parameter each epoch.

Possibly implement a **curriculum scheduler class**: it monitors progress (like validation accuracy on current distribution) and if above threshold, updates the data sampler to higher difficulty. This is a bit advanced but doable given we can measure e.g. token accuracy on validation for each difficulty bucket.

- **Multi-task balancing:** If one task is significantly easier, the model might master it quickly and we don’t want it to forget it while focusing on others. We’ll use **interleaved training** so all tasks remain in play. We might also occasionally freeze some part (not likely needed here). We will likely share the entire model for all tasks (with the slight difference in output handling for bracket vs others, but we unify that by treating “valid/invalid” as just output symbol as well with separate frequencies – the model can output that tone and we decode it).

- If multi-task proves too hard (the model weights conflict), an alternative is to train a separate model per task to optimal and then see if a combined model can be initialized from those or use knowledge distillation to a single model. This is complex; we hope multi-task works out since tasks share underlying needed skills (memory, logic).

Regularization: We will use dropout (maybe 0.1) in attention and possibly in SSM layers (some SSM implementations allow dropping certain connections). We have weight decay in AdamW. Given synthetic data is infinite, overfitting is mostly to patterns – negative controls will catch if it overfits logic. Dropout can be moderate to not hamper learning too much.

Logging and Checkpointing: - We will log training loss and breakdown (audio vs CTC vs others) every batch or so. - Use WandB or TensorBoard to visualize. Also log metrics on validation sets (exact match, token acc) periodically (maybe every 1000 steps). - Save checkpoints regularly (every certain number of steps or if new best on validation). Because training could diverge (especially if SSM goes unstable), frequent checkpointing helps to rollback. - We’ll keep best checkpoint for each task perhaps, but ultimately need one best for all tasks multi-task.

Training Duration: Possibly a few days of training on single 4070 for full tasks. But we plan to break it into stages; each stage might be a day or two. We’ll evaluate after each stage to decide to proceed or adjust.

Subject-Selector Implementation: In practice, this will be our custom data loader: e.g.

```
class MultiTaskDataset:
    def __init__(...):
        # hold sub-datasets for each task
    def __iter__():
        while True:
            task = choose_task(probabilities)
            sample = sample_from(task_dataset[task], difficulty=curr_diff[task])
            yield sample
```

We might not explicitly implement a reinforcement learning for subject selection (like some curricula do by difficulty), but manual scheduling is fine. We will update `curr_diff[task]` occasionally to raise difficulty.

We’ll also ensure each batch contains examples of only one task at a time (because output format and loss slightly differ, especially bracket output is different length). This simplifies computing loss (we know which criterion to apply). Alternatively, we could pad outputs to the same length (like treat “valid” as a single token output which is just 1 frame tone, similar enough to other tasks outputs being sequences). Actually, we can unify by always representing output as a sequence of symbols: - For bracket, output sequence length = 1 (just one symbol "V" or "X"). - For mod, output length = length of result number digits. - For copy, output length = input length (or double if repeating). So we can handle all tasks in one framework where each example has an output symbol sequence (just sometimes length=1). Then the audio output just corresponds to those symbols. That way our CTC head and decoding is uniform. We just need to expand the symbol vocabulary to include all symbols from all tasks (letters, digits, "%" and "valid"/"invalid"). The model can learn to output a "valid" token just like outputting any other symbol's tone. That’s elegant: treat "valid" as a symbol with a unique tone, same for "invalid". Then Task2 output is just that symbol’s tone. Similarly, "%" is an input symbol in Task3 with its tone.

This unification means we do not need separate heads or output arrangements for tasks; one model handles them. Loss calculation though: The audio loss compares waveforms which are different length per task. We can manage that by padding the shorter output waveforms or just computing loss up to their length. We'll do example-wise loss.

Therefore, Subject-Selector basically picks a task, generates an input-output audio pair, we run the model and compute losses accordingly (the loss functions themselves can handle any sequence length, thanks to CTC for symbol alignment and appropriate masking in STFT for padded portions).

Gradient accumulation and stability: If we see unstable training (especially because SSM might be sensitive), we will lower LR, ensure gradient norm clipping is on, and possibly accumulate smaller batches to keep updates smooth.

Evaluation during training: We will periodically evaluate on validation sets of each task (maybe every epoch or every few thousand steps) to see how each task is doing. If one lags, we might adjust its sampling weight or difficulty progression.

Early stopping: We aim to train until all tasks converge to high performance IID. We also keep an eye on OOD metrics (maybe test occasionally on OOD sets to gauge progress, though won't use them to tune hyperparams too much to avoid bias – but since OOD generalization is key, we might peek and adjust training to improve it, such as adding more training data variety or regularization if OOD is failing from overfit).

Potential issues and mitigation during training: - If the model has trouble converging on audio output (like outputs noise), we can increase weight on CTC early to encourage correct symbol output, then later reduce it to focus on audio fidelity. Possibly implement a schedule: e.g. start with heavy CTC weight (so it learns the sequence quickly), then after it achieves mostly correct sequences, increase audio loss weight to make waveform accurate. This is analogous to how one might first teach an idea then refine quality. - If the SSM part is hard to train (some SSM require careful initialization to be stable for long sequences), we might use tricks from literature like initialization of state matrices near identity or spectral normalization. The Mamba authors likely have defaults. We'll test a simple case (like copying short sequence) to ensure no explosion. - Attention: sliding attention doesn’ t have many knobs, but we need to ensure we mask properly and use relative positional bias or so if needed (maybe not necessary for such short context). - Logging: We will specifically log something like Exact match % and Symbol accuracy % on validation. For multi-step tasks, exact match means entire output sequence correct, symbol accuracy is per-symbol. This helps identify if errors are usually just one symbol off or many. - We also log edit distance average to see how close outputs are if not exact.

Selector adjustment example: If after some training, Task1 is near perfect but Task3 is lagging, we might increase probability of picking Task3 examples. Also possibly increase its relative loss weight (or simply feed more of it). The risk is forgetting Task1; to avoid that, still feed some Task1 occasionally (maybe freeze Task1 difficulty at high level so it always gets some rehearsal). We can also periodically evaluate Task1 to ensure it’ s retained; if not, do a mini fine-tune on it (or ensure multi-task training has it enough).

Thus, our training plan is adaptive: - Start single task to get baseline, - Then multi-task with careful sampling, - Increase difficulty as performance permits, - Use losses to guide correct output and fine-tune weights, - Always test generalization.

6) Evaluation & Diagnosis (Error Analysis, Failure Modes, Ablations)

For each task, we will conduct detailed evaluation to diagnose whether errors come from **reasoning failures** (the model got the logic wrong, outputting wrong symbol sequence) or **rendering failures** (the model had the correct symbol internally but the output audio was poor or mis-decoded by scorer). Our integrated scorer (or the auxiliary CTC outputs during training) can help in this analysis.

Evaluation Metrics: We will use: - **Exact Match (EM):** Percentage of examples where the entire output sequence (after decoding audio to symbols via the scorer) exactly matches the expected sequence. This is a stringent measure of success for tasks like copy and mod. For bracket (yes/no), it's essentially accuracy. - **Token Accuracy:** The proportion of output symbols that are correct and in the correct position, aggregated across sequence outputs. This helps gauge partial credit (e.g. if in a 5-symbol output, 4 are correct, token accuracy is 80%). - **Edit Distance:** The Levenshtein distance between predicted symbol sequence and target, possibly averaged or reported as an average distance or as a normalized score. This is useful if the model tends to insert or delete symbols. - We also specifically measure **Validity classification accuracy** for Task2, since that's binary, but that’ s same as EM in that case. - Additionally, **latency/efficiency** can be considered (though not main focus). We might measure average runtime per example, since the user hardware is modest.

We will evaluate on each relevant split: - **IID validation/test:** Expect high performance (ideally >95% or 99% EM if model converged). - **OOD-length:** We expect some drop; we'll measure how EM declines as length increases beyond train. This indicates extrapolation ability. - **OOD-symbol (Task1):** This is pass/fail essentially – ideally the model copies unseen symbols with near 100% success. If not, it means it overfit to specific symbol identities (bad sign). - **OOD-compose (Task3):** Evaluate success on two-step mod if only one-step mostly trained, or three-step if two-step trained. This tests if model can generalize operation composition. We’ ll measure accuracy for multi-step specifically.

Now, **Diagnosis of Errors:** We will categorize errors observed into: 1. **Symbol substitution errors (Reasoning):** The output sequence has wrong symbols (e.g. wrong remainder digit, or in copy task, a letter is incorrect or out of place). This implies the model’s logical computation or memory failed. For instance, in Task1, if one symbol is wrong or missing, it could mean memory capacity issue (dropped a symbol) or confusion between similar tones (maybe a rendering issue if it output something but scorer mis-identified it due to noise). 2. **Insertion/Deletion errors (Reasoning/Rendering):** E.g. output sequence too long or short. In copy, a deletion could mean model forgot a symbol (reasoning issue if memory overflow) or maybe output it too quietly so scorer missed it (render issue). In mod, an insertion would be outputting an extra digit (maybe it confused leading zeros or something). 3. **Timing/ segmentation errors (Rendering):** The model might output correct tones but not separated properly, causing the scorer to merge or split symbols incorrectly. For example, maybe it played two symbols too close so scorer thought it was one, or inserted too long a pause so scorer thought two symbols. These are rendering issues: the logic was right (the tones it intended were right) but the formatting in audio caused mis-decoding. We can detect these by listening or looking at spectrograms, or by comparing the internal symbol sequence (from auxiliary head perhaps) to the scorer-decoded sequence. If internal CTC (trained) says one thing but the final output decode says another, likely a rendering problem. 4. **Content (logic) errors:** For multi-step tasks, the model might output plausible but wrong result (like “4” instead of “5” for a mod, or mark a bracket sequence as valid when it’s not). This clearly indicates it didn’t fully learn the rule or had insufficient capacity. We’ll examine patterns: e.g., does it often fail on certain cases (like maybe always mistakes when A < B in mod, or when brackets have a certain structure)? This can direct us to specific fixes or curriculum improvements. 5. **Shortcut exploitation:** We should check if model might be using unintended cues. Negative control tests help here. For example, feed a bracket sequence with equal number of opens/closes but wrong order: if model predicts valid, it was probably using count as a shortcut. Or feed a copy task a sequence with a new symbol pitched differently: if fails, means it memorized frequencies. Or test mod with numbers beyond training range: if fails badly, possibly it never learned actual division logic, just memorized small arithmetic or pattern (like maybe it learned mod 10 by just picking last digit – which is a valid trick in decimal mod 10 actually, but if it learned mod 4 by looking at last 2 bits maybe that is actually a correct rule for base 10 mod 4? Actually in base 10, mod 4 depends on last 2 digits – if model exploited decimal properties without general concept, might still succeed).

We'll systematically generate specific **challenge sets** to test for these: - For bracket: as said, test balanced vs unbalanced with equal counts, test near-miss cases (like a sequence that is valid until the very last symbol which makes it invalid – see if model catches it). - For mod: test a set where A = B (should output 0 always, see if does), A < B (should output A, tests if it learned that rule explicitly), prime vs composite B differences, etc. - For copy: test new symbol, test extremely long sequence to see when it fails (maybe find the maximum length it can handle before memory fails).

Identifying Rendering vs Reasoning failures: - Use the **auxiliary symbol head outputs** during evaluation. If we keep the CTC head in inference (though normally we wouldn't), we can check what sequence it predicts internally. If the internal predicted sequence is correct but the actual output audio sequence is wrong, that's a rendering issue. If internal is also wrong, the model never got the logic. - Alternatively, compare the output audio decoded sequence to ground truth: - If it's completely off or systematically off (like always off by one in arithmetic), likely reasoning. - If it’s generally correct structure but occasional symbol differences (like in copy, maybe one letter is mistaken often by a similar frequency letter), might be rendering confusion or limited frequency resolution. - We can visually inspect some spectrograms of outputs vs targets to see if the right frequencies appear at right times. - For bracket, if it makes any false positive or false negative, that’s a logic error since output is just one bit – likely it mis-evaluated a case. We analyze those cases to see common pattern (e.g. maybe deep nesting beyond training).

We will compile an **error log** categorizing each error in test sets: e.g., "Task1 OOD-length: errors mostly starting after 2x train length, likely memory overflow in state" or "Task3 mistakes when second modulus is larger than first result," etc.

Ablation Experiments (at least 5): We will perform controlled experiments by removing or altering key components to validate their necessity: 1. **No Attention Ablation:** Remove the two attention layers entirely, using only 12 SSM layers. Train on the tasks (maybe a slightly smaller version due to more SSM layers to keep parameters similar). Evaluate how performance differs. Hypothesis: Without attention, model might struggle more with content-based operations (like maybe copying unseen symbol positions or focusing on a particular segment). If we see a big drop in, say, OOD symbol generalization or certain logic, it confirms attention gave a boost in content-based reasoning [1]. If performance is similar, perhaps SSM was sufficient (which is a result in Mamba paper, but in our multi-task scenario maybe not). 2. **No Auxiliary Symbol Loss Ablation:** Train a model with the same architecture and main loss but **without CTC/auxiliary loss**. Compare convergence speed and final accuracy. We expect this model to possibly reach correct outputs eventually (especially copy might still learn by pure audio loss), but likely slower or stuck in local minima. For example, it might output fuzzy tones or sometimes wrong results because it didn' t have that direct guidance. If the auxiliary loss was crucial, this ablation will have notably lower accuracy or require more epochs to catch up. This justifies our design of adding that loss. 3. **Different Input Representation:** We ablate the input entropy reduction method: - For one ablation, feed **raw waveform** frames to the model (perhaps using a simple linear downsampling or none at all, just frame at 16k or 8k resolution into SSM). This will extremely test the model (very long sequence, may be infeasible beyond short examples). We likely can only do this for very short sequences due to compute. But it will show if the model can handle raw or if training collapses due to length. Expect it to be much worse or not trainable due to long sequence and no explicit focusing on magnitude. - Another variant: use a fixed STFT front-end vs. learnable conv. If our main used mel, ablate by using conv or vice versa. See if a learned representation improves performance or not. For example, maybe the conv could learn something slightly better tailored, or maybe mel was already optimal. If performance doesn' t change much, mel was fine; if conv improves, it suggests maybe phase info or some subtle cues learned (but we try to avoid that). 4. **Change Downsampling Factor:** If we downsampled by factor e.g. 160 (to 100 Hz), try a shallower downsampling (50 Hz or so) vs deeper (200 Hz). With more downsampling, sequence shorter (good for model) but detail might be lost (maybe short tones become too short to resolve?). With less, model sees more steps (maybe too many). Evaluate which yields better accuracy. We expect there' s an optimal range. This ablation informs future scaling (like if tasks get more complex, how low can we go in representation). 5. **Curriculum vs No Curriculum:** We can ablate the curriculum strategy by training a model from scratch on the full difficulty distribution (all lengths mixed from start). Perhaps it will struggle or converge slower. We compare final performance or training curve. Likely curriculum helps especially for tasks like brackets (RNNs often fail to learn long bracket tasks if started with long sequences [12] [13]). 6. **State Size or Layer Count variations:** Not explicitly required, but we might do: - Use fewer SSM layers vs more attention layers (like 8 SSM + 4 attention) to see trade-off. Or a pure Transformer (0 SSM, all attention) to see if SSM is critical. We suspect pure transformer on these might overfit or handle differently, interesting to compare. - Use smaller state dimension N to see if memory capacity drops (e.g. N=16 vs 64). - Remove one of the two attention layers (e.g. keep only decoder cross-attn or only encoder local) to see which contributed more. For example, if removing cross-attn but keeping self-attn, does copy task break? If removing self-attn but have cross, maybe local feature extraction suffers. - Train tasks separately to see if specialized models perform much better than multitask. If separate is far better, our multitask approach might be compromising capacity. This would suggest perhaps using task-specific heads or weights would help.

For each ablation, we will measure the same metrics on the test sets. The results will be compiled: - e.g. a table of exact match on IID and OOD for main model vs ablations. We expect: - Without attention: possibly lower OOD generalization on tasks requiring content-based jumps (Mamba authors identified content reasoning improved with input-dependent SSM, but adding a bit of attention might still help). - Without auxiliary: likely more errors in complex tasks (especially mod or bracket might fail more often). - Without downsampling: possibly not even feasible for long inputs or runs out of memory, confirming necessity. - No curriculum: training likely takes much longer or fails to reach the same length generalization.

Interpreting results: - If an ablation doesn't degrade performance, that component might be unnecessary. E.g. if no-attention model performs equally, then attention wasn’t adding much (maybe SSM already handled content). - If performance drops significantly, that justifies the complexity we added (like CTC aux was needed, etc.).

Failure Diagnosis Process: When encountering a failed test case, we will: - Use the scorer to get predicted vs actual symbol seq. - If sequence is wrong, check internal CTC output if possible. - Inspect the input and output audio or patterns to hypothesize why. Possibly run a known algorithm (like for bracket, run a known DP) to see what the correct intermediate should have been and guess where model went wrong. - Example: If model output for a bracket sequence is "valid" when it’s invalid, check the prefix where it should have caught the error. Perhaps the model’s state saturates or resets incorrectly. We could attempt to probe model’s hidden state (like visualize the SSM state values over time to see if it correlates with stack count). If the SSM state dimension N is large, maybe one dimension might correlate with open count. We can do some correlation analysis: feed bracket sequences of varying structure and see if any hidden unit tracks the count. If not, model might not have learned counting properly. - For arithmetic, see if errors happen on certain numeric patterns (like always wrong when a carry is needed or when result has multiple digits).

We’ll use such insights to refine training if possible (maybe add data focusing on those cases, or adjust model).

Example error analysis: Suppose in mod tasks the model often outputs remainder off by a multiple of the divisor (like outputting A instead of A mod B when A < B as error). That indicates it didn’t learn the rule "if A<B, output A" and maybe was trying some weird mapping. We might address by adding more examples of that pattern with maybe a specific curriculum or even an auxiliary hint. But likely just noting it as a failure mode is enough for our research outcomes.

7) Risk Assessment (Top 10 Potential Failure Points & Mitigations)

We enumerate the most likely points of failure in the project and how to address them:

1. **Risk: Model fails to generalize and just memorizes training patterns (Overfitting/Shortcut).** For example, it might memorize the mod results for training pairs or memorize bracket patterns instead of learning rules. This would show up as high IID accuracy but poor OOD (especially OOD-symbol or OOD-length).
2. Mitigations:
    - Ensure training data is diverse (cover wide range of combinations so memorization is hard).
    - Include explicit **negative examples** and use OOD validation to catch this early. If we detect it, we can introduce regularization: e.g. apply **phase randomization** and **noise** to input so the model can’t memorize exact waveforms, forcing it to rely on relational features. We already do symbol variation to avoid direct waveform memorization (phase noise kills one common cheat path where model might just playback input for copy).

    - Use **negative control training**: e.g. occasionally feed inputs with random outputs (the model should not try to find pattern in those because there is none, it should just incur loss – if it tries to memorize them it wastes capacity).
    - Another approach is to intentionally leave gaps in training distribution (like not include certain combos) and see if model erroneously fills them (which means it’ s interpolating oddly). But primarily, robust OOD testing and then adjusting training if needed (like expand training distribution or add penalty for trivial solutions).
3. Engineering mitigations: monitoring training curves for signs of overfit (if training loss keeps dropping but OOD val flatlines).

4. Research mitigations: analyzing learned representations to ensure they encode general features (maybe visualize how a new symbol’ s embedding is handled).

5. **Risk: Model output audio is intelligible to the scorer (and auxiliary head) but not actually correct audio (Model cheats through auxiliary).** For instance, the model could learn to encode the sequence in some high-frequency pattern that the CTC head (since it’ s attached to hidden) can read, while the audible output might be garbled or off. Or it could produce an audio that fools the scorer but wouldn’ t fool a human (like using different frequency that is close enough that scorer mis-identifies).

6. Mitigations:
    - Design scorer to be strict and well-aligned with human interpretation (the scorer essentially is how we interpret, so if it’ s fooled, for our purposes that might still count as correct – but we want real correctness).
    - We can double-check outputs manually or with alternative decoding method if suspicious.
    - The phase randomization augmentation also prevents encoding information in subtle phase or ultrasonic signals, since we could simulate that noise in training.
    - If suspect internal cheating: we could temporarily remove the auxiliary and see if audio alone yields same output; if not, maybe the model was leaning on internal head.
    - Also, limit the capacity of the auxiliary classifier (just one linear layer) so it can’ t decode some complex hidden message unless that message is basically the tones themselves.
7. Engineering: The final evaluation is on audio output via scorer, so if the model fooled scorer consistently, that’ s a weird scenario (scorer is deterministic known mapping, not a learned model that can be fooled by adversarial examples easily; our tones mapping is straightforward, so model cannot find a different tone and still fool the frequency-based decoding because if it outputs a different frequency than expected for symbol, scorer will output a different symbol).

8. So risk of fooling scorer with wrong content is low if we design unique frequencies.

9. **Risk: Out-of-memory or computational slowdown due to sequence lengths or model size.** Perhaps the SSM or attention layers use more memory than anticipated for long sequences, or the 8GB GPU can’ t handle the batch size we want for the longest OOD case.

10. Mitigations:
    - We already plan heavy downsampling to keep lengths manageable (e.g. 100-200 steps per second rather than 16000).
    - We will use gradient checkpointing on the model layers if needed (trading compute for memory by recomputing layer activations in backward pass).
    - Reduce batch size for long sequences or do inference one by one for OOD tests if needed.
    - If still too high, consider splitting the input sequence for attention (which we do via local attention).

    - Use PyTorch memory profiling to identify any unexpectedly large allocations (like maybe the SSM implementation unrolls recurrent computation in a big tensor – if so, we might implement it iterative to save memory).
    - Worst case, limit the length of audio we feed (if a certain OOD length is impossible, note it as limitation and handle a smaller one).

11. Engineering solution: ensure code uses in-place ops or half precision to save memory. Also, if we run out, possibly move some parts to CPU (not ideal) or use multiple forward passes for segments (like process first half, then second, but that' s not straightforward for sequential).

12. **Risk: Training instability or divergence, especially for SSM layers.** SSM (especially if recurrent) can blow up if parameters make it unstable (like an eigenvalue >1 in state matrix). We might see loss go NaN or huge.

13. Mitigations:
    - Use **gradient clipping** from the start to catch any big gradient that could ruin weights.
    - Monitor loss – if it spikes, pause training, reduce LR by factor, maybe resume from earlier checkpoint.
    - Possibly constrain SSM parameters: e.g. initial A matrix can be set to have magnitude slightly <1 or use an initialization from Mamba paper that ensures stability. The Mamba code might have guidelines (like they might use specific parameterization to guarantee stability).
    - If still unstable, consider adding an L2 penalty on state magnitude or a normalization in SSM (some SSM variants include gating or normalization).
    - We can also shorten sequence or curriculum more gradually to not present very long sequences initially which could cause instability.
    - Use lower learning rate on SSM parameters specifically if needed (tune).

14. Engineering: incorporate try/except to catch NaNs and automatically reduce LR and resume.

15. **Risk: Insufficient model capacity or depth to solve hardest tasks.** With only 12 layers and small dimensions, maybe the model can't perfectly learn multi-step mod or deeply nested brackets. It might asymptote at some accuracy below 100%.

16. Mitigations:
    - We could increase state dimension N or d_model moderately if memory allows. Or allow a few more layers if needed (though we said 12 by spec).
    - Use the attention layers smartly – e.g. ensure cross-attn is present to help with tasks like copying long sequences, which might otherwise be very challenging for a fixed-size state to output arbitrarily long sequence (an RNN can, but must store entire sequence).
    - If one task is particularly straining capacity (likely Task3 multi-step or very long bracket), consider training a specialized model for it or adjusting multi-task weights to focus more learning capacity there.
    - Another approach: incorporate **feedback training** – e.g. break a difficult task into parts for the model. For mod, maybe teach it to output intermediate result as well (we could have asked it to output A mod B and then mod C, but we simplified to final – intermediate supervision could help if it's failing multi-step).
    - If capacity is the issue, that itself is a research finding (maybe need bigger model).
    - If absolutely needed, we can do gradient accumulation to simulate larger batch that might help model generalize better rather than capacity. But likely capacity is param count; only solution is bigger model or more layers.

17. If our 12-layer limit is firm, and it fails on e.g. 3-step mod, we might note that as a limitation and try to at least get 2-step working as partial success (maybe the user’ s expectation of "compose two-step success" is the initial goal, not necessarily 3-step).

18. **Risk: Task interference in multi-task learning.** The model might struggle to simultaneously learn three tasks with the same weights. It could manifest as one task’ s performance improving at expense of another (catastrophic forgetting) or oscillation where it can’ t optimize all objectives well.

19. Mitigations:
    - Use careful task scheduling: e.g. train tasks separately to near-convergence and then fine-tune together (so it starts from good points for each).
    - Possibly introduce **task-specific heads**: although we mostly unified outputs, we could still allow some specialization. For instance, maybe the bracket validity output tone is quite different from letter tones; the model might need different scaling – we could have a slight separate output layer for that task. But we tried to unify by just treating them as different symbols, which should be fine.
    - Loss weighting: ensure each task’ s loss is balanced. If one task’ s loss dominates (perhaps tasks with longer outputs have more contributions), we might normalize or weight by task. For example, bracket outputs only 1 symbol so its contribution to total loss might be smaller compared to copy which has many time frames of output. We may upweight bracket examples or losses to ensure it’ s learned.
    - Monitor each task’ s validation – if one is lagging drastically, increase its sampling frequency or loss weight in the multi-task mix.
    - Use **gradient blending** techniques if necessary (some research does separate gradient updates per task to avoid conflicts).
20. Engineering: can implement a simple cyclic schedule to ensure equal exposure, rather than random which might starve a task by chance.

21. If truly unable to reconcile, one pragmatic fallback is to maintain separate models per task (lower scientific value, but one could ensemble or something – probably not desired by user).

22. **Risk: Scorer errors or evaluation bias.** The scorer might incorrectly decode outputs, leading us to think the model is wrong when it’ s actually right (or vice versa). For example, if two symbol tones are too close in frequency, scorer might confuse them even if model output is technically a correct tone.

23. Mitigations:
    - Choose symbol frequencies far apart and use a robust decoding (maybe FFT with peak picking).
    - Test scorer thoroughly on clean data (like input generation itself fed to scorer should yield exactly the symbols).
    - If model output is noisy, maybe apply a slight filter or threshold in scorer (since in real evaluation we can allow some tolerance).
    - Also consider latency: if model doesn’ t put enough silence between symbols, scorer might merge them. We can tweak scorer to detect slight dips or known symbol length to separate. Or enforce in training that model includes small silences (maybe by design of data – we had put gaps, so presumably model will learn to include them because output loss will expect that pattern).

- We will double-check any surprising errors by visualizing the spectrogram to see if it was model or scorer. If it’ s scorer, we adjust scorer’ s algorithm (like window sizes or frequency resolution).

24. The scorer is not used in training (except indirectly via CTC which is internal), so at least it doesn’ t cause training issues. But for evaluation, we must ensure it's reliable.

25. **Risk: Timeline slips due to slow convergence or debugging overhead.** Training could take longer than expected or multiple runs to get right. With 8GB GPU, if we try too large batch or something, it might be slow.

26. Mitigations:

    - Use a smaller prototype to test everything (MVP on Task1, as planned) to iron out issues in pipeline. This reduces later debugging.
    - Profile performance: optimize bottlenecks (for example, if Python data generation is slow, we can generate data ahead or use numpy instead of pure Python).
    - If training is slow, consider using a lower sample rate or smaller model for initial experiments to get results faster, then scale up with confidence.
    - Keep the code modular so we can distribute tasks (like maybe run separate tasks training in parallel if had multiple GPUs or sessions – though user has one GPU, but we can sequentially fine-tune each).
    - The timeline in plan is somewhat aggressive (8 weeks for research is short); however, since data is synthetic and model is small, iteration cycle should be okay. We allocate specific tasks per week and aim for early results (like by week 4 we should have something publishable for at least simple OOD).

27. **Risk: SSM (Mamba) implementation issues or bugs.** If we implement from scratch or use new library, there might be hidden bugs or it might not integrate well with auto-diff. A bug could lead to wrong results or inefficiency.

28. Mitigations:

    - Use verified reference code if possible (the GitHub for Mamba or an official package).
    - Write unit tests for our SSM layer on a simple known sequence problem (like feeding an impulse and checking if it produces expected response).
    - If Mamba code is too complex to debug in timeframe, have a backup: e.g. use a simpler RNN or GRU to see if pipeline works, then swap in SSM.
    - Monitor training for anomalies specifically to SSM (like if gradients blow up or if output seems random, maybe SSM not working).
    - Also, ensure the two attention layers interplay with SSM correctly (sometimes mixing can be tricky if dimension mismatches or etc.).

29. **Risk: Output quality (audio fidelity) is too poor for certain symbols, causing errors.** This might be minor since fidelity is not primary, but if output tones are very noisy, the scorer might misread or humans might question if it solved it. E.g., maybe multi-resolution STFT isn't enough to ensure a clean tonal output, and the model outputs a fuzzy approximation.

    - Mitigations:
    - Increase weight on frequency domain loss to emphasize hitting the tone frequencies correctly.

- Possibly introduce an adversarial loss with a simple discriminator that checks if output sounds like the synthetic tone pattern (this might be overkill, and training GAN on these tasks might complicate stability).
- Alternatively, post-process model output by a small filter to clean it (we could do this since we only care about correctness, but that breaks end-to-end slightly).
- Or make model output directly the parameters of clean tones (but that returns to symbolic).
- Given our careful design (distinct frequencies, STFT loss), the output should be reasonably clean, so this risk is moderate.
- We'll verify by listening to some outputs; if they are too distorted, maybe the model had trouble generating high-frequency sine accurately. Possibly adding more capacity or fine-tuning on audio quality (maybe training a few more epochs with just audio loss once symbols are correct, to refine waveform) could help.

Each of these risks has been considered in our design. We will continuously monitor and apply these mitigations. We will maintain a **risk log** during the project, updating with any new issues encountered and how we solved them (this can be part of documentation in README).

8) Open-Source Plan (Repo Structure, Documentation, Reproducibility, Timeline for Release)

To ensure the project is reproducible and open for the community, we will set up a clear repository and plan early publication of results (even preliminary ones):

Repository Structure:

```
ProjectResonance-Jericho/
 ├─── README.md
 ├─── requirements.txt (or environment.yml)
 ├─── data/
 │    ├─── generate_task1.py (script to generate mirror data)
 │    ├─── generate_task2.py (generate brackets data)
 │    ├─── generate_task3.py (generate mod data)
 │    └─── ... (maybe example datasets or metadata)
 ├─── model/
 │    ├─── mamba_ssm.py (implementation of Mamba-2 SSD layer)
 │    ├─── attention.py (sliding attention layer implementation)
 │    ├─── mini_jmamba.py (definition of the full model combining encoder/decoder)
 │    └─── losses.py (CTC loss wrapper, STFT loss code, etc.)
 ├─── train.py (training loop script)
 ├─── evaluate.py (script to run evaluation on test sets)
 ├─── scorer.py (functions to decode audio to symbols for evaluation)
 ├─── utils.py (utilities for logging, scheduling curriculum, etc.)
 └─── experiments/
     ├─── config.yaml (example config for an experiment)
     ├─── run_task1_baseline.sh
     ├─── run_multitask.sh
     └─── ... (scripts for ablations or plotting)
```

We'll structure it so others can easily generate the same data and train the model. Possibly provide some pre-generated example data (though it's simple to generate on the fly). The code will be in Python (PyTorch for model).

README.md contents: - **Overview:** Problem description (end-to-end audio reasoning tasks) and highlight of results achieved (for example, "Our model solves audio copy and arithmetic tasks with high accuracy, demonstrating learned logic in audio domain"). - **Installation:** How to install dependencies (list packages like PyTorch, librosa or torchaudio for STFT, etc.). - **Usage:** How to run data generation (maybe not needed if code does it on the fly), how to start training (with example command-line or config). - **Model Description:** A brief explanation of the Mini-JMamba architecture and how to configure its parameters. - **Reproducing Experiments:** Step-by-step instructions: e.g. "Run `train.py --config configs/multitask.yaml` to train the model. Then run `evaluate.py` to get metrics on test splits. Pre-trained model weights for our best model are provided [link]." - **Results:** Present key results (table of IID vs OOD accuracies as claimed). Possibly include a small snippet of an example (e.g., an image of a spectrogram or a link to an audio file input vs output). - **Ablation & Analysis:** Summarize what we found from ablation (we can put detailed in a paper, but README can mention "Removing attention reduced OOD length accuracy by X%, confirming content-based attention aids generalization"). - **Contributing/ Open Issues:** Encourage others to test on more tasks or extend the code, etc.

Experiment Reproducibility: - Use fixed seeds for data generation and training (for consistent results). - Provide the exact configuration for each experiment in a config file or in code. - Possibly provide saved model checkpoints for the final model (so others can directly evaluate it). - Also provide some generated audio examples (maybe in a folder or via a notebook) so others can listen to the model’ s outputs and verify the correctness.

Open-Source Timeline (Milestones): - Week 1: Create the GitHub repo (private or placeholder public). Push initial code for data generation and a dummy model (maybe a simple baseline like identity) just to claim the idea. This serves as a timestamp “occupying a pit” as mentioned (占坑), ensuring we have an early presence. It might just have README with plan outline and the simplest demo (like copying a single tone). - Week 3: After MVP (Task1 solved), update the repo with the working code for Task1, and perhaps mark release v0.1 including example of audio copy working. This will be the first public result: e.g. a short audio clip in README where input says "ABC" in tones and output (model) echoes "ABC". We can also report metrics for Task1 (like 100% on length <=10, and maybe generalizing to length 15). - Week 5: After integrating tasks 2 and 3, push those features. Possibly release v0.2 including multi-task model code and maybe preliminary results (like bracket accuracy ~X, mod single-step ~Y). - Week 6-7: Once we have solid results (even if not perfect OOD, but at least something to show), prepare a **technical report or blog post** to accompany the code. Possibly write on arXiv or a project page detailing the approach and results. The README will link to that. - Week 8: Finalize documentation, clean up code (refactor if needed for clarity), and release v1.0 as open-source, which would coincide with possibly writing a short paper or submitting to a workshop. The README by then will highlight the key achieved results: - Example: "We achieved 99% accuracy in copying sequences up to length 10 and 95% for length 20 (unseen in training). The model can evaluate bracket sequences with 100% accuracy up to depth 5, and 90% at depth 8 (generalization). It solves single-step modulo with 100% accuracy for 3-digit numbers, and two-step modulo with ~90% accuracy on unseen combinations." - We will also clarify limitations discovered (e.g. maybe failing beyond certain length).

Ensuring replicability: We'll include the random seed in config and mention hardware specifics, but since it's a single GPU job, others with similar GPU should reproduce within margin.

Earliest Public Result: The user asked "what is the earliest result that can be made public (even if toy)?" - Likely by the end of **Week 2 or 3**, we can publicly show that Task1 (copy) works on a simple example including some OOD. That would be a concrete milestone. Perhaps a tweet or demonstration: input audio of "HELLO" in tones, output from model the same "HELLO" in tones, even if the model hasn' t done tasks 2 and 3 yet. - Another early result: maybe bracket short sequences by Week 3 or 4; we can show an example where model hears "(()" and outputs "invalid" tone. This would be compelling to show that end-to-end audio logic is happening. - So in summary: - **First toy demo (Week 3)**: End-to-end audio copy on new symbol/longer sequence (for example, train on A-E, test on a sequence containing F, model correctly echoes including F). Show waveform or spectrogram comparing input and output. - **Second demo (Week 4-5)**: Audio bracket checking e.g. input audio "(()" vs "()()", outputs correct tone. - **Third demo (Week 5-6)**: A two-step modulo example: input "13 % 5 % 3" in audio, output audio corresponding to "0" (since 13 mod 5 =3, 3 mod 3 = 0). That would illustrate multi-step reasoning.

We'll likely include such examples in the README or supplementary.

Finally, once we have the robust results, we will prepare a submission (if goal is to publish academically). The open-source repo can serve as artifact for that.

This plan ensures that from early on, we have code in a repository, updated incrementally, with clear documentation and reproducibility steps, culminating in a comprehensive release of code and (if possible) pretrained models. We will include in README the **timestamp of first commit** as proof of timeline (helpful for claiming novelty, etc.). Each commit/ release will be tagged with date, and maybe we also mention key progress in commit messages (like "Reached 90% OOD length accuracy on copy - commit date").

By structuring in this way, anyone can follow our progress week by week in the repo, and at the end replicate the experiments to verify the claims.

---

1 4 9 [2312.00752] Mamba: Linear-Time Sequence Modeling with Selective State Spaces
https://arxiv.org/abs/2312.00752

2 5 6 State Space Duality (Mamba-2) Part I - The Model | Goomba Lab
https://goombalab.github.io/blog/2024/mamba2-part1-model/

3 Audio Generation with Mamba using Determined AI - MLOps Community
https://mlops.community/audio-generation-with-mamba-using-determined-ai/

7 8 Mamba-2: Algorithms and Systems
https://pli.princeton.edu/blog/2024/mamba-2-algorithms-and-systems

10 state-spaces/mamba: Mamba SSM architecture - GitHub
https://github.com/state-spaces/mamba

11 UnivNet: A Neural Vocoder with Multi-Resolution Spectrogram Discriminators for High-Fidelity Waveform Generation
https://www.isca-archive.org/interspeech_2021/jang21_interspeech.pdf

12 13 aclanthology.org
https://aclanthology.org/2023.eacl-srw.15.pdf