ChatGPT

Plan for Scaling NeuroFMx into a Comprehensive Multimodal Foundation Model

Current NeuroFMx Model Architecture and Capabilities

NeuroFMx (Neural Foundation Model for population dynamics) is currently implemented as a selective state-space model with a multi-component architecture. The core **NeuroFMXComplete** model integrates the following stages [1]:

- **Tokenizers & Encoders:** modality-specific encoders (e.g. spikes, binned counts, LFP, calcium imaging) convert raw neural data into token sequences of dimension *d_model* [2] [3].
- **Mamba SSM Backbone:** a *Selective State-Space Model* (code-named “Mamba”) that processes long sequences with linear time complexity, supporting multi-rate temporal processing (multiple Δt scales) [4]. This backbone produces a sequence of latent features (shape *(batch, seq_len, d_model)*).
- **Perceiver-IO Fusion:** a set of latent query vectors (e.g. 128 latents of 512 dimensions by default) that attend across the backbone outputs [5]. This stage fuses information and reduces it to a fixed-size latent tensor *(batch, n_latents, latent_dim)* regardless of input length.
- **PopT Aggregator:** an optional *Population Transformer* module that aggregates across neurons in a **permutation-invariant** manner [6]. PopT (if enabled) learns population-level representations, aiding cross-session alignment by treating neuron identities abstractly [7].
- **Multi-Task Heads:** a collection of output heads for different training objectives [8]. Current heads include: a behavioral **Decoder** (predicting behavior from latents), a neural **Encoder** (reconstructing neural activity from latents), a **Contrastive** projection head (CEBRA-style latent alignment) [9] [10], and an optional **Forecast** head for future neural activity prediction [11]. These heads enable supervised, self-supervised, and predictive tasks simultaneously.
- **Transfer Adapters:** placeholders for **Unit-ID adapters** and **Session stitchers** to enable transfer learning across different neuron sets or recording sessions [12] [13]. For example, a UnitID adapter can map a new session’s neurons into the latent space of a pre-trained model by learning a small bottleneck layer while freezing the backbone [14].

In code, **NeuroFMXMultiTask** is a variant that runs all enabled heads in parallel for multi-task training [15] [16], making it easy to compute a composite loss. The design is modular and ready for multi-modal inputs – the project documentation highlights plans for additional encoders (calcium, two-photon imaging, etc.) and a fusion mechanism for multiple data streams [17] [18].

Overall, the model is intended to serve as a **foundation model for neuroscience data**, combining multiple *modalities* (spikes, LFP, calcium, behavior, etc.) and learning a rich latent representation that generalizes across **subjects, tasks, and even species** [19]. The inclusion of a contrastive head (inspired by **CEBRA**, a state-of-the-art contrastive embedding method [9]) is specifically to align latent factors with external variables (like behavior or task context), ensuring the learned manifold captures meaningful common structure across experiments.

Key takeaway: The current NeuroFMx codebase provides a solid architectural backbone (SSM + PerceiverIO) and multi-task learning setup. It is already equipped for multi-modal input and cross-session generalization (via PopT and adapters). These capabilities will be leveraged and extended in the plan moving forward.

Short-Term Plan: Multimodal Training Pilot on a $500 Cloud Budget

In the short term, we will conduct a focused training run in a cloud environment to **integrate multiple neural data modalities** and assess NeuroFMx's performance. With a budget of ~$500, we must optimize our choices of data and compute usage. This pilot will prioritize a subset of modalities and moderate model scaling to stay within budget, while still demonstrating cross-modal learning.

1. Data Acquisition and Preparation (Multimodal)

To cover a broad range of neural data, we should gather representative datasets from several modalities. Each modality will be processed into a consistent format (using NWB where possible, as the code supports NWB input [20]). We will focus on modalities that are most feasible given the budget and available open data:

- **Neuropixels Spiking Data (Multi-unit spikes):** Leverage large-scale electrophysiology recordings such as the **Allen Institute Neuropixels Visual Coding** dataset (multiple sessions of mouse visual cortex) and/or the **International Brain Laboratory (IBL)** dataset. These are available in NWB format or via SDKs (AllenSDK) [21]. We will use the provided NWB loader to bin spikes and extract simultaneous behavioral variables [22] [23]. For example, download ~20 sessions from Allen's Brain Observatory (as in the code's plan, ~395k sequences) [24], which should be manageable in storage. Use the `IBLDataset` and `AllenDataset` classes as needed to parse these files [25] [26].
- **LFP / ECoG Signals (Local Field Potentials or iEEG):** Many electrophysiology datasets also include continuous field potentials. For instance, the Allen Neuropixels data includes LFP recordings per probe. We can extract LFP channels from the same sessions, or use a public iEEG dataset (e.g., from the DANDI archive or another source). Prepare these signals by downsampling or filtering and use the provided `LFPTokenizer` (which applies a 1D conv or spectral encoder) [27]. If NWB files contain LFP in processing modules, the NWB loader can be extended to load that modality too.
- **Calcium Imaging (2-photon or widefield):** Incorporate neural imaging data such as the Allen Brain Observatory **2-Photon calcium imaging** sessions (e.g., Visual Coding 2P data) or other open calcium imaging datasets (possibly via DANDI). These provide ΔF/F traces for neurons or pixels over time. Use the `CalciumTokenizer` or similar, which presumably would bin or frame-average calcium traces into the model's token format [28]. If needed, create a simple tokenizer that takes a sequence of calcium fluorescence and produces a downsampled sequence of length matching the state-space model input length.
- **Behavioral and Stimulus Data:** Ensure each neural recording's accompanying behavior (e.g., running speed, eye position, lever presses, choices, etc.) is collected. These can be used as targets for the decoder head and for contrastive pairing. For visual neuroscience data, also collect **stimulus information** (e.g. images or trial descriptors). While we may not feed raw images into NeuroFMx in this pilot (due to computational cost), we can precompute stimulus feature vectors. For example, use a pre-trained vision model (like a ResNet or CLIP image encoder) to get feature embeddings of images shown to the animal. These features can serve as an additional input modality or for a contrastive objective aligning neural latents with stimulus embeddings (mimicking how CLIP aligns image and text – here neural activity and image). This step is optional for the $500 pilot if time/

compute allows; at minimum, keep trial/stimulus IDs to condition or evaluate the model's alignment to stimuli.

Data Download & Preparation: We will write dedicated scripts to automate downloading each dataset and converting it into the required format:

- For Allen Neuropixels: use the AllenSDK's `EcephysProjectCache` (as partially shown in the code) to download sessions to a local cache [29]. Then, convert each session's data to a compressed numpy or torch format (e.g. saving binned spike counts, etc. as done in the provided `StreamingNeuropixelsDataset` which expects `.npz` per session [30]). The repository's training script indicates a preprocessing step that produces these `.npz` files (likely by reading NWB and saving `spikes` arrays) – we should either use that or adapt the NWB loader to feed data on the fly.
- For IBL: use the ONE API (if available) or DANDI to fetch IBL NWB files. Then, use `IBLDataset` class to load spikes and behavior [25]. Save sequences similarly.
- For Allen 2-Photon: download NWB or NWB-converted files from the Allen Brain Observatory 2P dataset. Extract neural time series (dF/F of each recorded cell or ROI). If too large, possibly select a subset of sessions or temporal segments to stay within storage and compute limits.
- For any iEEG/LFP data: identify an open dataset (for example, **Neuropixels data includes LFP**, or use a smaller EEG dataset from EEG-BIDS or BNCI for variety). Preprocess by filtering (e.g. 1-100 Hz band) and downsample to, say, 250 Hz, then chunk into windows. Use or implement `LFPTokenizer` to produce low-dimensional features per window (perhaps via short-time Fourier transform or a small CNN as per design) [27].

Given the budget, it's wise to limit the total volume of data per modality in this initial run. For instance, we might use 20 sessions of spiking data (as the plan suggests) and perhaps 5-10 sessions of imaging data, rather than exhaustive entire datasets, to keep training time reasonable. Each session's data can yield many training sequences (the Allen Neuropixels example yields ~395k sequences from 20 sessions [24]). If we incorporate two or three modalities, we will sample from each during training (possibly alternating batches per modality or mixing them).

Modality balancing: We should structure the training data pipeline to either *combine modalities in each batch* (if the model can accept multimodal input simultaneously) or use separate batches for each modality type in round-robin fashion. The **simplest approach for now** (given the code's current state) is to train on one modality at a time per batch and use the contrastive/decoder losses to still guide common representation learning. For example, we can alternate batches: one batch of spike data (with behavioral targets) followed by one batch of calcium data (with possibly different targets), etc. This can be done by merging datasets or using a custom `IterableDataset` that yields from multiple sources.

2. Model Configuration and Training Strategy

For this pilot, we will use a slightly scaled-down model (to fit within budget and a single high-end GPU). Based on the configuration hints in the code and budget, a reasonable setting might be:

- **Model size:** use `d_model` around 256–384 (instead of the full 768 planned) and fewer backbone blocks (e.g. 4–8 Mamba blocks). The example config uses an even smaller size (128 dim, 4 blocks) for an 8GB GPU [31]. With a cloud GPU like an NVIDIA A100 (which we can afford for some hours under $500), we can likely go up to ~256d and ~8 blocks (~tens of millions of parameters). This balances capacity and cost. We keep `n_latents` ~64 and `latent_dim` ~256 for the PerceiverIO/PopT stages in this pilot.

- **Multi-task learning:** Enable the decoder, encoder, and contrastive heads during training (and optionally forecast if we have the data to support it). This means the loss will be a combination of: reconstruction loss (neural encoder head output vs input spikes/calcium), behavioral loss (decoder output vs measured behavior), and contrastive loss (InfoNCE style from the contrastive head). It's important to **balance these losses** so one task doesn't dominate. We can start with equal weighting or based on scale (e.g., if reconstruction MSE is ~1e-2 while contrastive is already scaled as logits, might be fine). We will monitor each component's loss. Using the **NeuroFMXMultiTask** class with `task='multi-task'` will yield a dict of outputs [32] [16], and we can compute losses for each. For example:
    - L_decoder = MSE or correlation loss for behavior (if predicting continuous positions or a classification loss if discrete behavior).
    - L_encoder = MSE for neural reconstruction (maybe averaged across all neurons).
    - L_contrastive = InfoNCE loss computed by selecting positive pairs (we can treat the *behavior* or *time proximity* as the positive signal as CEBRA does). Since code's `ContrastiveHead` likely produces projections, we may need to implement the actual contrastive loss in training loop (ensuring temporally close or same-condition samples have higher similarity [33]). If behavior labels are present, we can sample pairs with similar behavior as positive.
    - Optionally L_forecast if we predict future spikes: that would be another MSE between forecast head output and held-out future data.

The total loss could be a weighted sum: e.g. *L_total = L_decoder + L_encoder + 0.1 * L_contrastive* (giving contrastive a smaller weight initially if it's noisier). These weights can be tuned.

- **Training schedule:** Use a learning rate around 3e-4 with AdamW (as in the code config [34]) and train for perhaps ~50 epochs or until convergence on a validation set. We will set aside a validation portion of each dataset (e.g. 20%). Monitor metrics like decoding R^2, reconstruction error, and contrastive alignment (maybe measured via k-NN classification in latent space of behavior labels).
- **Compute environment:** With $500, we might afford roughly 100-200 GPU-hours (depending on instance pricing). For example, an AWS p3.2xlarge (V100 GPU) is ~$3/hour, or a p4d (A100) ~$12/hour. We could run ~40-50 hours on a powerful GPU. Training ~50 epochs on a moderate model with ~0.5M sequences might be borderline, so we may subsample sequences or limit each epoch to a certain number of batches (since 395k * 50 = ~20 million iterations which is too high). We can instead use **epoch** to mean one pass through a random subset (like 50k samples) and still get enough updates. Alternatively, use fewer sessions or shorter sequences. **Strategy:** iterate until loss convergence rather than strict epochs over full data; early stopping or a fixed budget of iterations (e.g. train on 1e6 samples total).
- **Performance optimization:** Enable mixed precision (AMP) to speed up and reduce memory [35]. Use data loading optimizations (multiple workers, caching). Since reading large NWB files repeatedly is slow, converting to numpy on disk (as done with `.npz`) and memory-mapping is useful [36] [37]. We should do that preprocessing offline (with the data download scripts).
- **Parallel modality training:** If possible within one model, consider *modality dropout* or alternating training: e.g. one training step on spiking data, next on calcium data. This can help the model not overfit to one type. Over time, the shared backbone learns to handle both. We must ensure the model knows which modality it's receiving, especially if data characteristics differ (spike counts vs calcium values). One simple approach: add a **modality token or embedding** at the input indicating modality type. For instance, prepend a learned embedding vector to the token sequence for "spike"

vs “calcium” before feeding into the backbone. This idea is analogous to how models use language or segment embeddings. This way, the model can adjust processing slightly per modality. (This isn’t implemented yet, so it’s a quick addition we’d implement).

- **Evaluation during pilot:** After training, evaluate on held-out test data for each modality:
    - Behavioral decoding accuracy (e.g. correlation or R^2 between predicted and true behavior signals).
    - Neural reconstruction error (can the latent code reconstruct held-out neural activity? e.g. measure per-neuron correlation).
    - If possible, qualitative check of latent space: e.g. use PCA or UMAP on the latent vectors to see if trials cluster by condition or behavior, indicating the contrastive learning is structuring the space.
    - Cross-modal generalization: if the model truly learns a shared representation, test that by training a simple linear probe on latents from one modality to predict outcomes on another modality. For example, take the latent space from calcium data and see if you can decode a variable that was only trained on spikes – any transfer would indicate shared representation. This is an advanced check but aligns with the goal of cross-domain features.

The expected outcome of this short-term experiment is to **demonstrate that NeuroFMx can ingest multiple neural data types and learn a unified latent representation** that is useful for various prediction tasks. With ~$500, we aim to see at least a significant performance on each task (e.g., decoding better than chance and reasonable reconstructions) and that including multiple modalities doesn’t break the model. We will also profile resource usage – e.g., ensure the model can run within memory and that ~50 hours of training yields convergence. This pilot will inform how to scale up further.

3. Short-Term Cloud Setup Notes

To maximize the $500 budget: - Use a spot instance or preemptible VM if possible for cheaper GPU time. - Save checkpoints regularly (e.g. every epoch or few hours) so that progress isn’t lost if using spot instances. - Use the configuration management (Hydra or similar) from the code to keep track of hyperparameters [38]. Since the codebase may already have a training script (`train_legacy.py`), adapt it for our multi-modal scenario or write a new training loop script that draws from multiple DataLoaders. - Leverage existing functions: The code’s `Config` in `train_legacy.py` already has many settings and possibly a one-cycle LR scheduler, etc. We can modify those values for our run [39] [34] (for example, increase `max_epochs` to an appropriate number for full run). - Monitor training closely for the first epoch or two: ensure losses are decreasing and no instabilities (the plan suggests checking for NaNs, gradient explosion, etc. [40]). - If one modality dominates (e.g. spike data is far more abundant than imaging), consider stratified sampling or oversample the smaller modality so the model doesn’t ignore it.

By the end of this short-term phase, we should have a trained model (or several variant models) that confirm the viability of multimodal training. We’ll also have all the necessary data pipelines and code infrastructure set up, which is crucial for the next phases.

Long-Term Vision: Towards a World-Changing Foundational Brain Model

With the pilot demonstrating feasibility, we can expand NeuroFMx into a truly comprehensive foundation model for neuroscience, given **no strict budget constraints**. In the long term, the goals include: **scaling**

up data (all relevant modalities and species), scaling up model capacity, improving architecture for multimodal fusion, aligning latent manifolds across domains, and enabling continual learning. Below, we detail how to achieve each aspect robustly and accurately using the latest methods (and some novel ideas).

1. Scaling Data Across Modalities, Tasks, and Species

To build a foundation model that captures *cross-species* and *cross-domain* knowledge, we need to train on an unprecedented breadth of data:

- **More Neural Modalities:** In addition to spikes, LFP, and calcium, incorporate **fMRI and EEG** data for human brain activity to extend across species. For example, include human fMRI datasets (like the Human Connectome Project for resting-state or task fMRI) and human/neurophysiological signals like MEG or EEG from open databases. These modalities operate at different timescales and spatial scales, but a powerful model can learn abstract correspondences. We might need new tokenizers (e.g., an fMRI tokenizer that divides the brain volume into regions or uses an autoencoder to compress images of brain activity). Including these ensures the model learns representations that bridge invasive animal data and non-invasive human data.
- **Multiple Species Ephys:** Gather electrophysiology from non-human primates (e.g., MIT's Reach Task dataset or NIH's NeuroNex data for monkeys) and other model organisms (zebrafish two-photon recordings, etc.). Having mouse, monkey, and human data will allow the model to find *common latent factors* that underlie neural computations across evolution. For instance, visual cortex responses in mice vs. primates could be aligned through the model's latent space if both see similar stimuli.
- **Diverse Tasks and Behaviors:** Include datasets spanning various behavioral paradigms – vision (e.g., responding to images/video), motor control, navigation, decision-making, memory tasks, etc. This could involve adding:
- **Neuropixels data from multiple brain regions** (Allen and IBL cover visual and frontal areas, but also include other tasks like decision tasks from IBL).
- **BCI datasets** where monkeys or humans make movements or computer cursor moves (to get motor cortex signals).
- **Audio and language tasks** (e.g., neural data from auditory cortex while listening to sounds or from speech experiments).

The aim is to expose NeuroFMx to as wide a variety of neural dynamics as possible, so it learns generalizable patterns. We will use the **same architecture** to process them, relying on the model to differentiate context via learned embeddings or context tokens.

- **Stimuli and Environment Context:** For each experiment, if available, include the external context (stimulus, environment state) as an auxiliary modality. For example, feed video frames or other sensor data through an **auxiliary encoder** (this could be a pre-trained CNN or transformer for that modality) and then into the NeuroFMx fusion module. By doing multimodal self-supervised learning (similar to how contrastive language-image models work), the model can align neural representations with the external world. This can greatly enrich the learned features (the model might learn, for instance, visual feature tuning, auditory tuning, etc., in its latent units). One novel idea is to perform **tri-modal contrastive learning**: neural data, behavior, and stimulus are all forced into a consistent latent space. Positive pairs could be (neural segment, corresponding video frame features) and (neural segment, corresponding behavior), etc., while negative pairs are mismatched.

This encourages the latent factors to represent the underlying cause of neural activity (the stimulus or intended behavior).

- **Quality and Consistency:** With unlimited compute, we can afford to be comprehensive: include hundreds of sessions and subjects. However, ensure data quality – for each modality, apply rigorous preprocessing (artifact removal, normalization). Convert everything into a unified format (e.g., NWB or our own standardized sequence files) so that our data loader can seamlessly load any sample from any dataset. We might maintain a **metadata index** that for each sample identifies its modality, species, brain region, etc., enabling stratified sampling or analysis.

As we scale up, we should take inspiration from large-scale successes in other domains: - In CV/NLP, foundation models train on massive diverse datasets. Here, our “internet-scale data” equivalent is aggregating many neuroscience datasets. Efforts like the **DANDI** archive and BRAIN Initiative data share can be tapped. Eventually, the model could be trained on the *union of dozens of datasets*, giving it a vast experience of brain activity patterns. - A recent work demonstrated a foundation model for mouse visual cortex that predicted novel stimuli responses by training on many recordings [41]. We aim to extend this idea across modalities and species, so the model becomes a universal predictor and feature extractor for neural data.

2. Model Architecture Enhancements for Multimodal & Cross-Species Scaling

With more data available, we can safely scale up the model size and add architectural components to fully realize the NeuroFMx-X vision outlined in the design document. Key enhancements include:

- **True Multi-Modal Integration:** Evolve the model to handle multiple modalities **concurrently** (not just one at a time). The documentation suggests a `MultiModalNeuroFMX` class where separate tokenizers process each modality and then a cross-modal attention module fuses them [18] [42]. We should implement this architecture:
- For example, if a dataset provides both spikes and LFP together (or spikes and imaging simultaneously), the model can accept a dict like `{"spike": spike_data, "lfp": lfp_data}` in forward. Each goes through its tokenizer to produce a sequence of d_model vectors. Then, use either **Perceiver-IO latents attending to each modality** or a simpler **cross-attention** that merges the sequences. The code plan shows an `nn.MultiheadAttention` to fuse modalities [43], which could be replaced or augmented by using the Perceiver’s latent queries to attend to all tokens from all modalities (perhaps by concatenating modalities along sequence dimension with modality-specific position encoding).
- Ensure the fusion mechanism can handle one or more missing modalities (sometimes we have only one modality present). Techniques like gating or learned modality embeddings can help the model not overly depend on one input. The benefit is a **cross-modal latent representation** where, say, if both spikes and LFP of the same event are present, the model can combine their information for a better latent encoding.

- **Adapters for Modality:** Consider adding small adapter networks specific to each modality before merging (similar to how Perceiver does modality-specific encoding). This could be as simple as a linear layer or as complex as a tiny transformer per modality that projects its features into a common space.

- **Increase Model Capacity:** With more compute, scale up dimensions to improve representational power:

- Increase state-space model size (e.g., d_model to 1024+, more SSM blocks, and more latent vectors). The Phase 3 of the plan suggests going from 128-d to 256-d etc. which bumped parameters to ~12M [44]. We can target even larger: perhaps 512-d with 16+ blocks (hundreds of millions of params). State-space models are linear in sequence length and quite memory efficient, so they can handle long sequences (which we'll need for long recordings).
- Increase the **multi-rate complexity**: use additional downsample rates (the plan suggests adding 8x, etc., to capture very slow dynamics alongside fast ones [45]). Multi-rate SSM could allow the model to simultaneously model e.g. 1 ms precision spikes and slow fluctuations over minutes.
- Enhance the **PopT** aggregator: If we incorporate extremely large populations or want better cross-session alignment, add more layers or a more sophisticated set-transformer in PopT [46]. We might also introduce *explicit normalization* across neurons (ensuring one session's neuron activations don't dominate) and even a learned alignment of neuron identity to functional space. For cross-species, a *species-aware PopT* could be beneficial – e.g., include a species embedding or separate aggregator per species that then merges.

- **Latent Diffusion generative model:** Introduce a generative modeling component as per the NeuroFM-X design [47]. Once the latent space is well-formed, train a diffusion model or an autoregressive model on the latents to generate synthetic neural activity or to perform imputation. This could be a separate model that takes NeuroFMx latents and learns to decode them into full neural time series (like a "decoder" model for generation). With unlimited compute, this is feasible and would allow simulations of neural activity for hypothesis testing. Latent diffusion could also help in data augmentation – generating semi-realistic neural data to fine-tune the model further.

- **Cross-Species & Domain Alignment Mechanisms:** We want the model's latent space to align similar neural patterns across species. Two strategies to achieve this:

- **Contrastive multi-domain training:** Extend the contrastive objective to push representations of different species' data closer if they correspond to analogous conditions. For instance, if both a mouse and a monkey saw the same visual pattern, ensure their neural embeddings for that stimulus are close. We can mine such pairs either through stimulus labels or unsupervised (perhaps via clustering stimulus features and matching across species). The contrastive head can be extended or we can add a secondary contrastive loss that uses species as a grouping variable (e.g., use positive pairs across species for same stimulus). This is akin to "alignment loss" used in multi-domain representation learning.
- **Domain adversarial training (DANN):** Introduce a discriminator on the latent space that tries to predict the species or dataset origin of a sample, and train the backbone to **fool** this discriminator (gradient reversal). This will encourage the core latent features to be invariant to species, retaining only task-relevant information. We have to be careful that some differences (like firing rate distribution differences) do not entirely vanish if they carry functional meaning, but the goal is to prevent trivial domain differences from fragmenting the latent space.

Additionally, use the **UnitID adapter** mechanism for cross-species if needed: for example, a human EEG has channels instead of neurons – a UnitID adapter could handle new channel mappings. The adapter idea is a powerful tool for **few-shot transfer**: freeze the main model and just train a small adapter for a new dataset. This was already anticipated for new sessions [14]; we can extend it to new species by giving the adapter more capacity if needed (maybe multi-layer). In long-term, this means the foundation model's core doesn't need retraining for every new dataset – you can deploy a pre-trained model and just learn a mapping for the new electrodes or measurement specifics.

- **Few-Shot Learning and Minimal Data Adaptation:** A world-changing model should enable users to train on minimal data from their specific experiment and still get great performance. To achieve this, we ensure the foundation model is richly pre-trained (as above). Then we provide easy adaptation pathways:
- The **LoRA (Low-Rank Adaptation)** on the last blocks, as mentioned in the design [48], is one method. We can keep 99% of weights frozen and just learn small rank updates for a new task. This drastically cuts the needed data and training time for fine-tuning.

- Another approach is **meta-learning** during pre-training: we could incorporate something like Model-Agnostic Meta-Learning (MAML) or a prompt-tuning approach such that the model learns to quickly adjust to new tasks. For example, during training we simulate "few-shot" scenarios by periodically training the model on a new small subset and forcing it to adapt fast (this is complex but potentially rewarding).

- **Continual Learning Design:** While the initial focus is offline batch training, we want the architecture to eventually support continual learning (updating with new data streams without catastrophic forgetting). We can design the training pipeline and model with this in mind:

- Use replay buffers or generative replay: since we plan a generative model, the model could generate pseudo-data from past tasks to not forget them when learning new tasks (a known strategy in continual learning).
- Modularize the model per domain: e.g., have different adapter modules for different data streams that can be turned on or off, rather than one model that has to fit everything at once. The core knowledge remains in the backbone, but new experiences are integrated via new adapters rather than overwriting core weights.
- Regularization-based CL: apply penalties like EWC (Elastic Weight Consolidation) or synaptic intelligence on the backbone weights so that when fine-tuning on new data, it doesn't drastically change weights important for old data.

- Monitoring drift: in an online setting, continuously monitor if new data's loss starts increasing for old tasks, and if so, trigger a rehearsal or lower learning rate, etc. These are implementation details for the future online learning phase.

- **Robustness and Accuracy:** As we scale, also implement best practices:

- **Hyperparameter tuning** at scale (perhaps using the provided hyperparameter search script or an AutoML approach), to find the optimal learning rates, weightings for the multi-task losses, etc., for the massive training.
- **Regularizations** like dropout (the plan even suggests increasing dropout if overfitting [49]), batch or layer normalization (already used in heads [50]), and perhaps stochastic depth in deep stacks to stabilize training.
- If training on extremely long sequences (millions of timesteps), consider dividing into manageable chunks or using the state-space model's ability to handle streaming (some SSMs can be stateful so you don't have to feed the entire sequence at once). This could tie into an eventual online learning setup.

In summary, an ideal long-term NeuroFMx will have a **flexible multimodal encoder (spikes, LFP, imaging, fMRI, etc.), a powerful temporal backbone (SSM) that can model dynamics from milliseconds to hours, and a set of heads that not only decode and predict but also generate and align**. Its latent space will act as a common currency for neural information: neurons to behavior, one species to another, one modality to another – all can be related via transformations in this shared space.

3. Training at Scale: Strategy for Unlimited Compute

With essentially no budget limit, we can employ strategies akin to those used in training large language models or Vision Transformers:

- **Distributed training**: Use many GPUs (multi-node if necessary) to handle the load of data. Data parallelism will allow feeding different recordings in parallel. We need to ensure that the PyTorch Lightning or custom training loop we use supports this. Possibly switch to PyTorch Lightning Trainer for multi-GPU training to simplify scaling.
- **Curriculum learning**: Start the training on simpler or smaller-scale data, then gradually introduce more complex or high-dimensional data. For example, first train on spike trains (which are relatively sparse signals) to get the backbone weights in a good state, then introduce imaging data which has different noise characteristics, then fMRI which has lower time resolution but spatial info. This curriculum can help the model not get confused early on by very different distributions.
- **Combined objective and staged training**: We might stage the training objectives as well. For instance, first train a purely self-supervised model (using reconstruction + contrastive) on all data to learn a good latent space without focusing on any one supervised task. Then, once the backbone is robust, add the supervised decoder head objectives (behavior decoding etc.) and train a bit more (or train them jointly but perhaps with lower weight initially). This ensures that the model doesn't overfit to easily-supervised signals at the expense of general representation quality.
- **Benchmarking and iteration**: The plan outlines a comprehensive benchmarking suite [51] [52]. We will evaluate the foundation model against known methods like **CEBRA, LFADS, and Neural Data Transformers (NDT)** on standard tasks [53]. For example, compare decoding accuracy on a held-out dataset or latent space quality metrics. These benchmarks will guide us to adjust the model (if, say, we find the latent space has lower clustering quality than CEBRA, we increase weight on contrastive loss or add more latent dimensions).
- **Monitoring and fail-safes**: At the scale of unlimited training, it's important to monitor for divergence or mode collapse in generative parts. Set up regular validation checks and have the ability to backtrack to earlier checkpoints if something goes awry (much like "testing phases" in the plan [40]).
- **Documentation and open-source**: As this becomes a world-class model, documenting the process, hyperparams, and publishing results will be valuable. This includes sharing the final pre-trained model weights so others can use it (model registry and serving infrastructure is already considered in the plan [54] [55]).

Finally, **continual learning integration**: once the offline model is strong, we can begin deploying it in real-time environments (labs, brain-computer interfaces) where it continuously receives new data. Start with pseudo-online updates (e.g., fine-tune on new session data after each session, simulating online). Evaluate if performance on prior data drops; if so, incorporate the CL strategies mentioned (replay, adapters, etc.). Over time, this could evolve into a system that **never stops learning**, akin to how an AI like GPT continually ingests new text. The difference is our model would ingest new neural recordings, constantly refining its understanding of brain activity.

Actionable Implementation Steps (for the Development Pipeline)

Below is a **set of detailed instructions** that an AI coding assistant (e.g. "Claude code") can follow to implement the next steps in parallel. These tasks cover data acquisition, code development for multi-modality, and training orchestration. Each step can be undertaken independently to speed up progress:

1. **Data Download & Preparation Scripts** – *for each new modality:*
2. **Neuropixels Spiking (Allen & IBL)**: Write a Python script to download the Allen Institute Visual Coding Neuropixels data. Use the `allensdk.brain_observatory.ecephys` API to fetch session data into a `./data/allen_neuropixels` directory (as expected by the code) [56]. Include options to download a specified number of sessions (configurable, e.g., 20 sessions). After downloading NWB files, the script should convert them into our training format: open each NWB, extract spike times and units, bin them at 10 ms (as NWBDataset does [57] [23]), and save an `.npz` file per session containing arrays `spikes` (shape (num_sequences, seq_length, num_units)) and `behavior` (aligned behavioral data). Do the same for IBL: use the ONE API or download NWB from FlatIron, then process similarly (the structure may differ, ensure to extract e.g. wheel movements or choices as behavior).
3. **Calcium Imaging (2P)**: Create a script `download_allen_2p.py` to download Allen Brain Observatory 2-Photon data (or an alternative public calcium dataset). For Allen 2P, the AllenSDK has an API for that as well. Download experiment NWB files for a set of sessions (across different visual areas and stimuli for variety). Then, implement a conversion: extract fluorescence traces for all recorded cells. Likely downsample these to, say, 10 Hz to reduce sequence length, and chunk into segments of length ~100 time points. Save an `.npz` or similar with `calcium` array (segments x time x cells). If available, also store a `behavior` array (e.g., running speed or pupil size) and `stimulus` identifiers per segment. This script should output data files ready to be loaded by a new `CalciumDataset` class (which we will implement if not already).
4. **LFP/iEEG**: Write a script to fetch an LFP dataset. For example, extend the Allen Neuropixels script to also extract LFP data from the NWB (Allen provides LFP in a separate file per probe). Alternatively, use an iEEG dataset from DANDI (like UCLA seizure dataset or similar). The script should filter and resample the raw signals to a manageable sampling rate (maybe 250 Hz), then split into windows of fixed length (e.g., 1s windows = 250 samples). Save as `.npz` with `lfp` array (windows x time x channels). If there are behavioral markers (like whether the subject is doing a task or resting), include those as well.
5. **fMRI (Optional)**: If including fMRI, write a script to download a sample (like a few subjects from HCP or an open mouse fMRI dataset). Preprocess by parcellating the brain into regions (to reduce dimensionality) and treat each time series as a “unit”. Then chunk into sequences. Save `fmri` data similarly. This is optional and can be a later addition, but structuring the code for it now will ease future integration.

Each of these data scripts should be designed to run in parallel. They can be separate processes or threads since they access different sources. Ensure each script writes into a structured folder (e.g., `data/allen_neuropixels/processed/`, `data/allen_2p/processed/`, etc.) so that downstream code can discover available datasets.

1. **Extend Data Loading Code for New Modalities**:
2. Implement new `Dataset` classes or extend `NWBDataset` to handle the processed files for calcium, LFP, etc. For instance, create `CalciumDataset` similar to

`StreamingNeuropixelsDataset`, which loads `.npz` files of calcium segments. Ensure each dataset's `__getitem__` returns a dictionary with keys matching what the model expects (e.g., `'spikes'` or `'calcium'` as the neural data, `'behavior'` for any behavioral target, etc.). If using NWBDataset for all, you might integrate modality as a parameter and handle accordingly.

3. Modify the collate functions or dataset classes such that we can identify the modality of each sample. Possibly add a field `'modality'` in the returned dict or use different keys for spikes vs calcium.

4. Test these dataset classes with a small sample to ensure they load data correctly (you can unit test by printing shapes, etc.). This will likely be done with dummy data if real data is large.

5. **Implement Multi-Modal Model Support**:

6. Create a class `MultiModalNeuroFMX` (if not already in code) that inherits from `NeuroFMXComplete` [58]. In its `__init__`, instantiate one tokenizer per modality needed (spike, LFP, calcium, etc.) using the existing tokenizer classes [59]. Each tokenizer should output sequences of shape (B, S, d_model). If some tokenizers output different lengths (due to different sampling rates), you might upsample or downsample within the tokenizer to a common sequence length or handle it in fusion.
7. Implement the `forward` such that it accepts a dictionary of modality -> data. For each modality present, pass the raw data through the corresponding tokenizer to get tokens. Then combine these token sequences:
   - Option 1: Concatenate along the sequence dimension (e.g., treat them as parallel sequences). This might require padding shorter sequences or truncating longer ones. You could also interleave them if timepoints align, but if rates differ, that's complex.
   - Option 2: Use a cross-modal attention: e.g., have a set of latent queries (like Perceiver latents) attend jointly to all modalities. The plan shows a placeholder using `nn.MultiheadAttention` [43]. A possible implementation: concatenate modality token sequences along *feature* dimension or stack them and use attention to produce one fused sequence. Alternatively, you can simply concatenate the token sequences and feed through the existing backbone as if it were one sequence (but then backbone must handle potentially larger sequence length and mixed content).
   - Given we already have PerceiverIO in the pipeline, an elegant approach: feed each modality's tokens *separately through the Mamba backbone* to produce modality-specific features (or even skip backbone per modality and directly use Perceiver). Then have the PerceiverIO stage attend to all modalities combined. However, this is a significant change. A simpler near-term implementation is to perform an early fusion: combine tokens then backbone as is. This should work if we include modality embeddings to distinguish them.
8. Ensure that after fusion, we pass the fused tokens into the rest of the pipeline (backbone → perceiver → etc.). If we did early fusion pre-backbone, then just call `super().forward(fused_tokens)` as in the pseudo-code [60].
9. Add a capability for missing modalities: e.g., if a modality is not provided, either skip its tokenizer or provide zeros. The model should not crash if one modality is absent. Testing tip: try forward with only one modality in the dict.

10. **Modality Token/Embedding:** Implement a simple mechanism to inform the model of modalities. For example, add a learned embedding vector for each modality type and **add** it to that modality's token embeddings. This can be done by maintaining a small lookup (e.g.,

`self.modality_embeddings = nn.ParameterDict` with entries for 'spike', 'lfp', etc., each a vector of length d_model). When tokenizing, after `tokens = tokenizer(data)`, do `tokens = tokens + modality_emb[modality]`. This way, the backbone can differentiate sources.

11. **Unified Training Loop for Multi-Modal Multi-Task Learning**:

12. Develop a new training script (or modify the existing one) to handle multiple DataLoaders and the `MultiModalNeuroFMX` model:
    - Initialize the `MultiModalNeuroFMX` with appropriate parameters (d_model, etc. as decided). Enable all relevant heads (`enable_decoder=True`, etc.) and set `encoder_output_dim` equal to the number of neurons for spike modalities (and similarly for others, if using reconstruction on them).
    - Create DataLoaders for each dataset (spike, calcium, etc.), with shuffling. If data sizes differ widely, you might set their `epoch_length` or number of iterations proportionally so one modality isn't oversampled.
    - Use a training loop that iterates over batches from each DataLoader in turn. One approach: Python's `zip_longest` to cycle through, or manually alternate. For each batch:
    - Prepare an input dict for the model. For example, if the batch comes from spikes dataset, you might get batch = {'spikes': ..., 'behavior': ...}. If from calcium dataset, batch = {'calcium': ..., 'behavior': ...}. Convert that to a dict matching model forward: e.g. `input_dict = {'spike': batch['spikes'], 'lfp': batch['lfp'], ...}` – basically include whatever modalities are present in the batch. (Batches from one modality will have only that key, which is fine.)
    - Pass `input_dict` through model: `outputs = model(input_dict, task='multi-task')` using the multi-task forward (this will output a dict of heads).
    - Compute losses: If `outputs` has keys 'decoder','encoder','contrastive', compare each against batch targets. For decoder, target might be `batch['behavior']` (make sure shapes align; if decoder predicted one step or sequence). For encoder (neural reconstruction), target is `batch['neural']` or `batch['spikes']` depending on how dataset returns it (the NWB loader returns `'neural': same as spikes` [61]). For contrastive, the `outputs['contrastive']` might be an embedding vector; you need to form positive/negative pairs. Implement an InfoNCE: e.g., take the output embedding of current sample and one other sample from the same batch that has a similar behavior value (if batch has behavior labels, identify positives by threshold on similarity or by time adjacency if sequential). If behavior labels are continuous, an approach is to treat each sample's nearest neighbor in behavior space as positive (this is how CEBRA does using behavioral similarity). Simplest: for each sample i in batch, pick sample i+1 as positive (since data likely sequential), and others as negatives. Compute contrastive loss with temperature.
    - Sum the losses (with weights if needed) to get total loss. Backpropagate.
    - Optimize with AdamW. Use gradient accumulation if batches must be small due to memory, as in config (they did 8 steps accumulation) [62] .
    - Continue this training loop for the desired number of iterations or epochs. Optionally use a learning rate scheduler (OneCycle or cosine as plan suggests) [63] .
    - Include periodic validation: have a small held-out set for each modality, and perhaps evaluate a few key metrics every epoch.
    - Save checkpoints periodically (especially the best one by validation score).

13. This training loop might be easier to implement using PyTorch Lightning by creating a LightningModule that handles multi-modal data, but given our custom approach, a manual loop is fine.

14. **Evaluation and Benchmark Scripts**:

15. Implement code to evaluate the trained model on various tasks: decoding accuracy, reconstruction error, latent space analysis. For instance, a script `evaluate_model.py` that loads a checkpoint and runs the `NeuroFMXBenchmark` suite as outlined in the plan [51] [64]. This includes functions to compute MSE and correlation for reconstruction, R^2 for decoding, and some measure of latent space geometry (you could compute average between-condition distance or use a pre-written function). Automate this to output a summary of performance.
16. Also implement a small routine to test cross-domain performance: e.g., take the model trained on multi-modal data, freeze it, and train a linear classifier on the latent features for some new small dataset (few-shot transfer test). This corresponds to the benchmark_transfer_learning mentioned [65]. This will give insight into how well the representation works with minimal data (aligning with our goal of few-shot learning).

17. Compare results with baselines if possible: e.g., run a simpler model or previously published result on the same evaluation data. This can be a separate analysis, but prepare code to run a baseline (maybe a basic LSTM or PCA followed by decoder) for reference.

18. **Future Continuous Learning Setup (Design Only for Now)**:

19. (This is more of a design task than coding immediately.) Outline how one would integrate an online learning loop. Possibly create a skeleton `continual_training.py` that simulates receiving new data periodically and updating the model with an adapter or limited training, then tests on a memory of old tasks to check for forgetting. We don't execute this now, but having the scaffold will guide future development. Include placeholders for replay buffer or saving model state after each incremental update.

Each of these steps can be executed in parallel to accelerate progress. For example, while data scripts are downloading datasets (#1), you can concurrently start modifying the model code for multi-modal support (#3). After data is prepared, the training loop (#4) can be run. The evaluation (#5) can be written while the model is training (to be ready when training finishes).

By following these implementation steps, we will build the infrastructure needed to **scale NeuroFMx to a true foundation model**. The outcome will be a system capable of training on diverse neuroscience data, capturing shared representations across modalities and species, and adaptable to new scenarios with minimal effort. This not only sets the stage for high performance on current benchmarks but also paves the way for **continual learning** from streaming data in the future, inching us closer to a continually learning AI neuroscientist.

---

[1] [2] [3] [5] [7] [8] [12] [13] [14] [28] neurofmx_complete.py
https://github.com/sidhulyalkar/neurOS-v1/blob/e30af5733bd79f242daff50ecfd6ab12ef9ab556/packages/neuros-neurofm/src/neuros_neurofm/models/neurofmx_complete.py

4 6 17 19 27 38 47 48 NEUROFM_X_PLAN.md
https://github.com/sidhulyalkar/neurOS-v1/blob/e30af5733bd79f242daff50ecfd6ab12ef9ab556/docs/NEUROFM_X_PLAN.md

9 10 11 33 50 heads.py
https://github.com/sidhulyalkar/neurOS-v1/blob/e30af5733bd79f242daff50ecfd6ab12ef9ab556/packages/neuros-neurofm/src/neuros_neurofm/models/heads.py

15 16 32 neurofmx_multitask.py
https://github.com/sidhulyalkar/neurOS-v1/blob/e30af5733bd79f242daff50ecfd6ab12ef9ab556/packages/neuros-neurofm/src/neuros_neurofm/models/neurofmx_multitask.py

18 24 31 36 37 40 42 43 44 45 46 49 51 52 53 54 55 58 59 60 63 64 65
NEUROFM_FOUNDATION_MODEL_PLAN.md
https://github.com/sidhulyalkar/neurOS-v1/blob/e30af5733bd79f242daff50ecfd6ab12ef9ab556/packages/neuros-neurofm/docs/NEUROFM_FOUNDATION_MODEL_PLAN.md

20 22 23 25 26 57 61 nwb_loader.py
https://github.com/sidhulyalkar/neurOS-v1/blob/e30af5733bd79f242daff50ecfd6ab12ef9ab556/packages/neuros-neurofm/src/neuros_neurofm/datasets/nwb_loader.py

21 29 30 34 35 39 56 62 train_legacy.py
https://github.com/sidhulyalkar/neurOS-v1/blob/e30af5733bd79f242daff50ecfd6ab12ef9ab556/packages/neuros-neurofm/training/train_legacy.py

41 Foundation model of neural activity predicts response to new ...
https://www.nature.com/articles/s41586-025-08829-y