ChatGPT

Evaluation of the NeurOS-v1 BCI Platform

Introduction and Overview

NeurOS-v1 is a comprehensive open-source platform for real-time neural data processing, designed as a modular "operating system" for brain-computer interface (BCI) research. It provides end-to-end capabilities – from data acquisition and signal processing to machine learning inference and user interfaces – all within a single Python-based framework. The system is currently at version 2.0.0 (Beta), with core functionality in place (single- and multi-modal pipelines, CLI, ~10 models, ~15 drivers) but still under active development [1]. NeurOS emphasizes clarity and extensibility: it reimagines an earlier codebase with modern design patterns (asynchronous agents, pluggable modules, standard data formats) to deliver a production-oriented BCI toolkit. In this evaluation, we examine NeurOS's code design, implementation, and features in detail, compare it to existing BCI tools, and discuss its strengths, weaknesses, and architectural choices (monolithic vs. modular). The goal is to assess NeurOS-v1 as one would for a top-tier conference/journal submission – i.e. its novelty, engineering quality, use-case coverage, and how it could be improved or better packaged for the community.

Architecture and Code Design

Agent-Based Modular Architecture: NeurOS is built around an **agent-based** pipeline orchestrator that cleanly separates data acquisition, signal processing, and model inference into distinct asynchronous components [2]. A central **Orchestrator** agent coordinates three core sub-agents – a **DeviceAgent** for data input, a **ProcessingAgent** for filtering/feature-extraction, and a **ModelAgent** for classification or prediction – connected by in-memory queues. This design enforces clear separation of concerns: each agent has a focused responsibility and communicates via well-defined interfaces (queues), which simplifies reasoning about the pipeline. Importantly, the entire pipeline leverages Python's `asyncio` for concurrency, enabling real-time streaming and parallel processing with relatively low overhead. The orchestrator can spawn these agents as tasks, run them concurrently, and collect metrics like throughput and latency in real time [3] [4]. This asynchronous, event-driven design is a **modern approach** to BCI systems, contrasting with older frameworks that often used sequential or threaded pipelines. It allows NeurOS to achieve low end-to-end latencies (on the order of milliseconds) and high sample throughput, which is crucial for responsive BCIs.

Modularity and Extensibility: The codebase is organized as a **single Python package (`neuros`)** with clear sub-modules for each component (e.g. `drivers`, `processing`, `models`, `agents`, etc.), and uses abstract base classes to enforce consistent interfaces. For example, all data sources subclass a `BaseDriver` interface, implementing an asynchronous generator for timestamped data samples [5] [6]. This means any new hardware device or data modality can be added by simply extending `BaseDriver` and writing a `_stream()` method – the rest of the pipeline doesn't need modification to accommodate it. Similarly, models derive from a common `BaseModel` (and for advanced cases, `BaseFoundationModel`), ensuring they expose standard methods (`train()`, `predict()`, etc.) and can be hot-swapped. Filters and feature extractors are also pluggable: the pipeline can accept a list of filter objects or a custom processing agent at runtime. NeurOS even supports **auto-configuration**: given a high-level task description (e.g. “motor imagery with EEG”), it can auto-select an appropriate driver, model, and filter band parameters [7] – a valuable feature to lower the barrier for new users. Overall, the code design follows modern Python best practices (type hints, dataclasses, f-strings, rich docstrings) and appears to be **well-documented and readable**, which speaks to its clarity and suitability for an open-source research tool [8] [9].

Production-Oriented Features: Beyond the core pipeline, NeurOS includes architectural touches aimed at robustness and deployment. For instance, a **QualityMonitor** can be attached to the processing stage to monitor signal health (e.g. SNR or artifact levels) and adapt the pipeline if data quality degrades. An **AdaptiveThreshold** mechanism can adjust decision thresholds on the fly to maintain stable performance [10] – useful in non-stationary environments common in BCI (e.g. drifting baselines or fatigue). The orchestrator and agents have been built with fault-tolerance in mind: the DeviceAgent/Orchestrator can gracefully fall back to a simulated driver if real hardware fails (e.g. if the BrainFlow library for EEG hardware isn’t present, NeurOS will issue a warning and use a MockDriver instead of crashing). This attention to graceful degradation and system health is a strong design choice for a “platform” intended to run continuously in experimental or clinical settings. Furthermore, NeurOS provides hooks for **observability** and external integration: the multi-modal demo pipeline exposes Prometheus metrics (so you can monitor performance externally), and there’s support for publishing data to Kafka streams in real-time [11]. These features, uncommon in academic BCI code, indicate that NeurOS is engineered with real-world use and **scalability** in mind (e.g. integrating with cloud data pipelines or dashboarding systems).

In summary, the architecture of NeurOS-v1 is **clean and well-structured**. It successfully balances **flexibility** (via modular, plugin-like components) with **integration** (a unified orchestrator ties everything together). Such design makes the codebase not only easier to extend (new modules can plug in without refactoring core logic) but also easier to test components in isolation. One notable drawback is that the current integration tests are incomplete – only ~40% of tests were passing at audit time [12] – which suggests that while the design is solid, more work is needed to validate and harden the implementation across all scenarios. Nonetheless, from a code design perspective, NeurOS demonstrates a high engineering standard appropriate for a tool aiming at **NeurIPS-level** quality: it’s **innovative** (applying modern software architecture to BCI) and **open for extension**, which is critical for a research platform that must evolve with new algorithms and devices.

Data Acquisition and Drivers

One of NeurOS’s strongest features is its extensive support for **data acquisition from various modalities**. The framework includes **15+ drivers** (at beta release) covering a wide range of signal sources: EEG and other electrophysiology (with BrainFlow integration for many EEG amplifiers), peripheral biosignals like EMG, EOG, ECG, GSR (skin conductance), respiration, and even non-physiological streams such as audio input, video cameras, motion sensors (IMUs), and more [13] [14]. Each driver is implemented as a subclass of the base driver interface, meaning they all provide a common asynchronous stream of data (as NumPy arrays with timestamps). This unified API for real or simulated hardware is a significant improvement in usability: researchers can **“hot-swap”** drivers without changing the rest of their code. For example, one can prototype an EEG pipeline using a `MockDriver` (simulated data) or a file-based replay driver, and later switch to a `BrainFlowDriver` to stream from an actual EEG headset – no changes needed in the processing or modeling code. This is analogous to hardware abstraction layers in an operating system, and NeurOS handles it elegantly.

Compared to other BCI platforms, NeurOS's driver layer is exceptionally broad. Traditional academic tools often focus only on EEG: e.g., **BCI2000** (a classic C++ BCI framework) supports EEG hardware and some aux channels, but adding new device types requires custom C++ modules. **MNE-Python**, while great for offline EEG/MEG data analysis, does not offer built-in real-time acquisition (it can read from EEG hardware via LabStreamingLayer, but it's not its core strength) [15]. NeurOS not only covers EEG but also integrates emerging modalities like **fNIRS/HD-DOT** (optical imaging), which are rarely supported in unified frameworks. The **multi-modal sync** capability in NeurOS's *Constellation* pipeline highlights this: it can ingest and time-synchronize **diverse data streams (EEG, audio, video, EDA, fNIRS, respiration, etc.) simultaneously** for a holistic experiment [11]. Such built-in multi-modal support is cutting-edge – it goes beyond what tools like **Braindecode** (deep learning toolbox for EEG) or **MNE** offer, and even beyond BCI2000's typical use (which often handles one modality at a time or assumes manual synchronization externally).

In terms of implementation, drivers like the EEG BrainFlow driver or an OpenBCI driver likely use external SDKs (BrainFlow, pylsl) to interface with hardware. The NeurOS design smartly makes these dependencies **optional**: e.g., if `brainflow` Python package or the LabStreamingLayer library isn't installed, the system will warn and degrade to a dummy driver [16] [17]. This ensures that missing a hardware dependency doesn't break the whole system – a nice touch for portability. Each driver runs in the asynchronous event loop and pushes data into a queue with a configurable buffer (with backpressure – the queue drops old samples if a consumer falls behind [18]). This is important for real-time performance: it prevents memory bloat if, say, the processing step briefly can't keep up with a high data rate. Overall, the driver module in NeurOS is well-designed and **comprehensive**. It fits into the project's mission by making the **input stage extremely flexible** – accommodating many use cases from replaying recorded data to live streaming from multiple biosensors. Few BCI frameworks achieve this level of input generality natively, giving NeurOS an edge in experiments that involve more than just EEG.

Signal Processing and Feature Extraction

NeurOS pipelines include a configurable **signal processing stage** that can be tailored to the experiment's needs. By default, for EEG data the system uses a **band-pass filter** (to isolate relevant frequency bands) followed by a smoothing filter and a **bandpower feature extractor**, which computes feature vectors (e.g. power in delta/theta/alpha bands) from the raw signal [19]. However, this default can be easily changed: the user can supply a list of custom filter objects or even replace the entire processing agent with a custom class. NeurOS's processing module implements common DSP operations: e.g., filtering (band-pass, notch), artifact removal (not detailed in the excerpt but likely included or planned), and augmentation techniques. The mention of "8 EEG-specific augmentation techniques" in documentation [20] suggests that NeurOS has utilities for data augmentation (important for training robust models on limited BCI data).

This pluggable processing pipeline is similar in spirit to how **MNE-Python** or **EEGLAB** (Matlab) let users apply various filters and transformations – but NeurOS does it in a streaming, real-time context. That means these operations are implemented as asynchronous code processing chunks of data as they arrive. The *ProcessingAgent* takes data from the driver's queue, applies the sequence of filters, then passes results into a features queue for the model. The **design choice to dynamically load processing "plugins"** (via specifying filter classes in config or CLI) is excellent for extensibility: new signal processing methods (e.g., a wavelet transform or a different feature like Hjorth parameters) can be added as new classes and used without altering core code. This is much cleaner than monolithic pipelines where adding a new step requires editing a long script.

Importantly, NeurOS also considers **multi-modal processing**. In the *Constellation* multi-modal demo, each modality can have its own processing chain and feature extraction before fusion. For modalities beyond EEG, the system allows a custom ProcessingAgent (for example, processing video frames or motion data might require different logic than filtering EEG). This flexibility is crucial: a one-size-fits-all processing pipeline would not work across EEG vs. video vs. motion. NeurOS handles it by allowing modality-specific processing classes to be plugged in [21] [22]. The orchestrator will use the default EEG pipeline unless a different agent class is provided for a given stream. This design ensures that each data type is processed appropriately, yet all modalities still funnel into a unified model in the end.

From a **code quality** perspective, the processing code benefits from the structured approach (each filter likely a class with a consistent interface). The use of numpy/SciPy for filters and feature extraction means reliability and speed via vectorized operations. One minor concern might be performance with Python-level code if very high-frequency data or many channels are used (Python loops could become a bottleneck). However, the asynchronous nature and the ability to offload heavy computation to numpy (which is in C) or even PyTorch/TensorFlow (for learned features) mitigates this. In any case, the framework's focus seems to be on moderate sampling rates (EEG ~250 Hz by default [23]) which are easily handled in Python.

In summary, NeurOS's processing module is **well thought-out and flexible**. It fits perfectly into the whole project by acting as the bridge between raw signals and machine learning, and it can be adapted to various research needs. This is superior to many older BCI systems where processing is hard-coded or limited to a fixed sequence. In NeurOS, the pipeline can evolve with new algorithms – a valuable trait for a platform aiming to stay current with scientific advances.

Machine Learning Models and Classification

NeurOS comes with an extensive **model suite** built-in, covering both classical machine learning and deep learning approaches to neural data decoding. According to the documentation, over **10 models** are implemented, ranging from simple classifiers (logistic regression, SVM, k-NN, random forests) to neural network models like a 1D CNN, an LSTM, a Transformer, and EEG-specific models like **EEGNet** [24]. This breadth means that out-of-the-box, researchers can try a variety of algorithms on their data without having to write model code from scratch – a big convenience for benchmarking. Each model adheres to a common interface (`BaseModel`), which typically defines methods like `train(X, y)`, `predict(X)`, and possibly `save/load`. This uniform API makes the models **interchangeable** in the pipeline: one can swap a RandomForestModel for an EEGNetModel in the config, and the pipeline code doesn't change except for that instantiation. It's a strong design for experimental flexibility.

Beyond the standard models, a standout feature in NeurOS is its integration of **"foundation models" for neural data** – large-scale or state-of-the-art models drawn from the latest research. Notably, NeurOS includes wrappers for models like **POYO/POYO+** and **NDT3**, which are cutting-edge multi-session neural decoders described in recent NeurIPS and ICLR papers [25]. It also provides an implementation of **CEBRA** (a technique for latent neural embeddings) and a so-called **Neuroformer** (a transformer-based zero-shot learning model) as per the documentation [26] [27]. These are **unique selling points** of NeurOS – to our knowledge, no other open-source BCI toolkit has incorporated such advanced models directly into its library. For example, **Braindecode** focuses on deep learning for EEG, but mainly provides standard convolutional networks (like variants of EEGNet or shallow/deep ConvNets) and doesn't have these recent transformer-based or multi-task models built in. **MNE-Python** and others don't include learning models at all (they rely on users bringing scikit-learn or TensorFlow themselves). By including foundation models, NeurOS is trying to bridge the gap between classical BCI methods and the latest machine learning research – which is exactly the kind of ambition that would appeal to a NeurIPS audience.

The implementation of these advanced models is handled via wrappers. For instance, the POYOModel in NeurOS will attempt to import an external package `torch_brain` (which presumably contains the reference implementation of POYO) and if available, use it; if not, it logs a warning and falls back to a stub [16] [28]. This optional dependency approach is wise given not all users will need or want to install heavy libraries. It appears NeurOS does not re-implement these foundation models from scratch (which is good, as that'd be a huge effort), but provides integration so they fit the BaseModel interface. In practice, a researcher could load a pre-trained foundation model (`model = POYOPlusModel.from_pretrained(...)`) and use it within the NeurOS pipeline seamlessly [29]. This enables things like **transfer learning** (applying a model pre-trained on one dataset to your own with minimal fine-tuning) and **few-shot or zero-shot decoding**, which are cutting-edge capabilities in BCI. For example, NeuroformerModel's zero-shot prediction API lets you feed a task description and get predictions without explicit training for that task [30] – an exciting prospect if it works as advertised.

In terms of how these models fit the whole project: having a "model zoo" inside NeurOS makes the platform a **one-stop shop** for BCI algorithm evaluation. You can compare classical approaches to deep learning to foundation models under a unified evaluation pipeline. The CLI even supports an offline **training mode** (`neuros train`) to train a model on recorded features, and then a **run** mode for real-time inference, which aligns with how one would typically use these models (train offline, then deploy online). This integration of training and deployment is valuable; some frameworks only handle one or the other. On the downside, packaging so many models does bloat the codebase and dependency list – NeurOS ends up depending on heavy libraries like PyTorch, scikit-learn, etc., making installation a bit cumbersome. There's also the question of whether all these model implementations are fully tested and up-to-date. The audit suggests model saving/persistence is still in progress (no robust model registry or versioning yet) [31], and some foundation models might be placeholders awaiting their reference code. So, there is complexity in maintaining a large model suite.

Nonetheless, in an academic evaluation, the **breadth of NeurOS's model support is a major positive**. It indicates the tool's capability to serve as a **benchmarking platform**: one can fairly compare methods on the same data with the same preprocessing and metrics. And it demonstrates forward-thinking design by incorporating the latest research (foundation models) early. To improve further, the project should continue fleshing out model persistence (so models can be saved/loaded easily between sessions) and perhaps provide more examples or tutorials on using each model (since such complex models can have non-trivial usage or tuning requirements). Comparatively, NeurOS's integrated model zoo puts it ahead of most existing BCI software in terms of out-of-the-box algorithms available.

Orchestration and Real-Time Performance

At the heart of NeurOS is its **Pipeline Orchestrator** – effectively the runtime engine that links drivers → processing → models into a live loop. This orchestrator is implemented as an async agent (subclass of a

BaseAgent) that launches the other agents and supervises the run [3] [4]. The orchestration logic is designed for real-time operation and does a few important things to ensure performance and stability:

- **Concurrent Streaming:** Because the device, processing, and model agents run as separate asyncio tasks, they can operate in parallel on different batches of data. For instance, while the ProcessingAgent is extracting features from batch *n*, the DeviceAgent might already be reading batch *n+1* from the device. This pipelining yields low latency. The documentation notes that typical inference latencies can be <10 ms [32], which is plausible given the concurrency and if using efficient models. In comparison, older single-threaded loops (common in simple BCI scripts) would accumulate more latency since each step waits for the previous one to finish.

- **Timing and Duration Control:** The orchestrator can run for a specified duration or indefinitely until stopped [33]. It uses wall-clock time to determine when to halt, ensuring experiments run for the intended length. This is important for standardized benchmarking (e.g. run *exactly* 60 seconds of data). NeurOS leverages asyncio's sleep for timing, which should be accurate enough for these purposes (though high-precision timing might depend on event loop characteristics, likely fine here).

- **Metrics Collection:** As data flows through, the orchestrator collects metrics such as the count of samples processed, and individual sample latencies (time from input to output) via a callback on the ModelAgent's results [34]. After the run, it computes throughput (samples/sec) and average latency and returns these in a metrics dictionary [35]. This built-in **benchmarking instrumentation** is extremely useful. It allows researchers to quantify performance of the pipeline easily. The CLI `neuros benchmark` command likely leverages this to output a JSON report of latency/throughput/accuracy [36] [37]. This focus on quantitative evaluation is something we'd expect in a high-quality research tool, facilitating fair comparisons between NeurOS and other systems or between different configurations.

- **Multi-Modal Fusion:** In the case of multiple simultaneous data streams, NeurOS provides a specialized **MultiModalOrchestrator** (and corresponding agents) to handle synchronizing and fusing features from each modality before feeding them into a single model. The *Constellation* demo is a prime example: it starts multiple DeviceAgents (one per modality), then either uses multiple ProcessingAgents or a composite agent, and finally a ModelAgent that receives combined feature vectors. The orchestrator ensures all these are launched and stopped together, and likely handles alignment (possibly using timestamps to sync modalities). This is a complex task (synchronization of heterogeneous data), but NeurOS's design suggests it can manage it within one framework – an ambitious and powerful capability not found in most simpler BCI pipelines.

In terms of reliability and testing, the orchestrator and agents are core to everything, so their correctness is critical. The fact that the basic pipeline tests were passing (the audit noted `test_pipeline_run_returns_metrics` passed [38]) is a good sign. However, more complex orchestrations (e.g. with real hardware or multiple streams) were not fully tested yet (the constellation command was noted as untested in the audit [39]). This indicates a need for more integration testing in future development, especially if NeurOS is to be used in sensitive scenarios (e.g. clinical trials or long-running experiments). From a code perspective, one potential improvement might be better error handling or graceful degradation in the orchestrator: for example, if the ProcessingAgent throws an exception (perhaps due to unexpected data), does the orchestrator stop everything cleanly and report the error? Ensuring robust error propagation and shutdown is key in real-time systems to avoid deadlocks or zombie tasks. Given the asynchronous design, adding timeouts or monitors to detect stalled loops could also be valuable (though there's no indication of issues, these are general considerations).

To compare with other systems: **BCI2000** also has the concept of sources, filters, and outputs (it calls them modules), but those are compiled and communicate via shared memory in a controller GUI – a very different (and less flexible) architecture than NeurOS's dynamic Python agents. NeurOS's orchestrator is arguably more developer-friendly and transparent (you can inspect or modify the Python code easily). **OpenViBE** (another BCI platform with a graphical designer) can do multi-threaded pipelines, but adding new algorithms in OpenViBE is not trivial (requires C++ plugins). NeurOS's pure-Python orchestrator with asyncio is much easier to extend and integrate with other Python code (like adding a custom callback to log data to a database is a few lines in NeurOS, whereas in closed frameworks it could be impossible or hard). So in terms of *real-time pipeline control*, NeurOS is at the level of the best research platforms and offers a more accessible environment to the Python-savvy scientific community.

User Interface and Deployment Tools

NeurOS provides multiple ways for users to interact with and deploy the BCI pipeline, which enhances its usability in different scenarios:

- **Command-Line Interface (CLI):** The package installs a `neuros` CLI tool that aggregates various convenient commands. For example, `neuros run` launches a live pipeline with specified parameters (device, duration, etc.), printing results and metrics to the console. `neuros benchmark` runs a standardized evaluation and outputs performance metrics as a report [40] [37]. There are also commands for training models offline (`neuros train` expects a dataset of features/labels to fit a model), for launching the Streamlit dashboard (`neuros dashboard`), for running multiple predefined tasks (`neuros run-tasks`), and even for generating Jupyter notebooks (`neuros demo`) to demonstrate usage. This CLI is well-designed for **quick experimentation and debugging** – a user can get started without writing any code, or a lab instructor could use it to demonstrate BCI concepts interactively. Notably, NeurOS's CLI stands out compared to most academic toolkits: MNE-Python has no unified CLI, Braindecode doesn't either; BCI2000 does have a GUI/CLI but it's specific to its environment. The presence of a CLI in NeurOS shows attention to the researcher's workflow (scriptable experiments, automation, etc.).

- **Dashboard (Streamlit):** For a more visual, real-time monitoring experience, NeurOS offers an optional **Streamlit-based dashboard**. With `neuros dashboard`, it can spin up a web app that likely displays incoming signals, model outputs, and possibly system metrics in real time. Streamlit is an easy way to make GUIs for Python code, so this is a smart choice. It allows running experiments and seeing results (plots of the EEG signals, classification probabilities, etc.) without manually plotting. This could be very useful for demonstrations or for monitoring long experiments (e.g. seeing if signal quality drops, etc.). Competing tools: BCI2000 has its own GUI windows for signal monitoring, but those are not easily customizable. MNE can plot raw data or EEG topographies but not in a unified dashboard way. So NeurOS having a web-app style dashboard is a plus for user experience. The downside is Streamlit can add overhead and complexity; but since it's optional (not installed unless needed), it doesn't bloat the core.

- **REST API and WebSocket Streaming:** Impressively, NeurOS includes a **FastAPI-based server** (`neuros serve`) that exposes the pipeline functionality via RESTful endpoints and WebSocket streams [39]. This means you can start a NeurOS pipeline as a background service and interact with it from other applications or over the network – for example, sending a request to start/stop a run or retrieving real-time data/predictions via a WebSocket. The fact that it even has authentication and token-based security built-in [41] shows it's geared towards production deployment or multi-user scenarios. This is quite beyond typical research code; it positions NeurOS as something that could be deployed on a server for, say, a cloud BCI service or a lab facility accessible to multiple users. Competing frameworks rarely have this: to get a REST interface on MNE or Braindecode, you'd have to wrap it yourself. NeurOS providing it out-of-the-box is a **strong differentiator** if one aims to integrate BCI functionality into larger systems (e.g. a web app that needs to display brain data in real time).

- **Data Logging and Formats:** On the data side, NeurOS supports writing data in standard neuroscience formats like **NWB (Neurodata Without Borders)** and following the **BIDS** conventions [11] [42]. This is important for interoperability and publication: NWB is an increasingly adopted format for sharing raw and processed neurophysiology data. By using NWB (and also Zarr and WebDataset for certain outputs), NeurOS ensures that data collected can be easily shared or analyzed with other tools. For example, one could record a session with NeurOS, save it as NWB, and then use MNE-Python or MATLAB to do further analysis on the saved data. This use of standards is commendable and relatively novel (many BCI platforms historically had custom data formats). Additionally, NeurOS has an internal SQLite-based logging for events/metrics and can export curated datasets to WebDataset shards for machine learning pipelines – features that show its aim for **end-to-end workflow coverage** (from data acquisition to dataset creation and model training).

- **Deployment Support:** While not fully complete, NeurOS includes some deployment aids like a Docker Compose setup (for running the Kafka stack and presumably the NeurOS services) [43]. The audit notes no Kubernetes manifests or cloud-specific instructions yet [44], but those are on the roadmap. This indicates that the maintainers are considering how someone might deploy NeurOS in different environments (local, cloud, cluster). For an academic project, having any Docker support is a plus because it helps with reproducibility (e.g. you can distribute a container for a NeurOS-based experiment). As this matures, NeurOS could become a truly turnkey solution (one-click deploy of a full BCI system with data streaming, model serving, and dashboards).

Overall, the **interface and deployment modules** of NeurOS greatly enhance the core pipeline. They make the system accessible to different users: neuroscientists who are not coders can use the CLI or dashboard; developers can integrate via the API; and engineers can deploy it as part of larger systems. In comparison to other tools, NeurOS clearly outshines most in this category – many frameworks focus only on algorithmic aspects and leave interface as an afterthought. Here it's integrated. The only caution is that these features (dashboard, API, etc.) add to the complexity and require maintenance. The audit showed some tests failing around the API security (authentication issues in tests) [45], meaning these advanced features need further polish. But conceptually, they are a big plus, aligning the project with real-world usage where a BCI system isn't just run in a notebook, but perhaps needs to be **user-friendly and network-accessible**.

Comparison to Existing BCI Tools

NeurOS-v1 enters an ecosystem with a variety of BCI software frameworks, each with different focus areas. Here we compare NeurOS to a few notable tools to gauge its relative strengths:

- **MNE-Python:** *MNE* is a well-known Python library for EEG/MEG data analysis. It excels at offline signal processing (filtering, epoching, spectral analysis, source localization) and visualization, but it is not designed for real-time closed-loop use. MNE does interface with real-time streams to some extent (for example, through the LabStreamingLayer and FieldTrip buffer), but it lacks a built-in orchestration for streaming pipelines or any machine learning model integrations. By contrast, **NeurOS provides real-time streaming and ML out-of-the-box**, which MNE doesn't [15]. MNE also doesn't include classification algorithms internally (users typically use scikit-learn or external libraries on MNE-processed data), whereas NeurOS includes a whole suite of classifiers and deep models. On the other hand, MNE is very mature and has rich visualization and signal processing utilities (topographies, artifact rejection, etc.) that NeurOS might not fully match. One could envision NeurOS and MNE being complementary: e.g., NeurOS for running an experiment in real-time, then exporting data to MNE for detailed analysis. NeurOS adopting standards like NWB/BIDS makes this feasible. Summing up, NeurOS's scope is **broader (real-time + ML)**, but MNE is more **established for analysis**. For a NeurIPS-like evaluation, NeurOS would be seen as pushing beyond MNE by enabling new capabilities (foundation models, multi-modal streaming) that MNE doesn't attempt.

- **BCI2000:** *BCI2000* is a classic general-purpose BCI system (originating circa early 2000s) with a modular design (source, signal processing, output) and a GUI for configuration. It's highly optimized for real-time performance and has been used in many BCI research studies. However, BCI2000 is written in C++ with some MATLAB support; it's not easily extensible by average users – adding a new algorithm often means diving into C++ code. NeurOS, being pure Python, offers a much more **accessible and extensible** framework for modern researchers. In terms of functionality, NeurOS and BCI2000 share goals: both can do real-time filtering and classification of EEG. BCI2000 has some features like stimulus presentation integration and a user interface for tasks, which NeurOS doesn't explicitly mention (NeurOS is more focused on the data pipeline, not stimulus control). But NeurOS leaps ahead by integrating **deep learning models** and multi-modal data, which BCI2000 (focused mostly on EEG) lacks. BCI2000 also doesn't natively support things like sending data to web dashboards or using cloud infrastructure – it's a standalone system. NeurOS's design is more in line with current software ecosystems (APIs, data formats, etc.). One could say NeurOS is a **next-generation equivalent** of what BCI2000 was: aiming to be general-purpose, but with modern tech (Python async vs. C++ threads, web integration vs. local GUI). For a high-level review, NeurOS would be praised for overcoming limitations of BCI2000 (closed architecture, difficulty of adding new models) by using a more open and modular approach. The trade-off is that BCI2000 is very stable (decades of use), whereas NeurOS is new and not yet battle-tested in the field to the same degree.

- **Braindecode:** *Braindecode* is a Python library specifically for deep learning on EEG/MEG. It provides implementations of networks like EEGNet and deep convnets, and utilities to train them on datasets (often leveraging MNE for data I/O). Braindecode's focus is offline model training for classification/regression tasks; it does not handle real-time streaming or data acquisition – one typically uses it on recorded datasets. NeurOS overlaps with Braindecode in the sense that both have deep learning models for EEG. In fact, NeurOS implements EEGNet and a Transformer model, similar to what Braindecode offers. However, NeurOS wraps those models into a real-time pipeline context (you can deploy them on streaming data easily), which Braindecode doesn't directly support. Also, NeurOS extends beyond EEG to other modalities and includes non-deep models, whereas Braindecode is purely focused on neural networks. In terms of code design, Braindecode leverages PyTorch and is integrated with the Braindecode/MNE ecosystem; NeurOS uses PyTorch too for its deep models but has a lot more infrastructure around them. If we benchmark, say, training an EEGNet on a dataset, Braindecode might be a bit more user-friendly (since that's its sole purpose, with tutorials for training). NeurOS can also train EEGNet but perhaps with a bit more configuration (NeurOS might need the user to call `model.train(X, y)` or use its CLI, etc.). A reviewer might note that NeurOS re-implements some functionality that could be gotten from Braindecode/MNE (like dataset loading, certain transforms), which could be redundant – but at the same time, NeurOS's philosophy is to provide a unified platform. So, NeurOS stands out by offering **both training and deployment** in one place, whereas Braindecode is more limited in scope. If NeurOS is to compete as a toolkit, it effectively subsumes some of Braindecode's territory and then goes further (by being real-time capable).

- **MetaBCI:** *MetaBCI* is a recently released open-source platform (2023) that, like NeurOS, aims to cover the entire BCI pipeline in Python. It includes modules for dataset handling (integrating many public EEG datasets), preprocessing, and online experiment pipelines. MetaBCI's structure is somewhat modular as well (with sub-packages named "brainda", "brainflow", "brainstim" for different purposes) [46]. The goals are very aligned: both MetaBCI and NeurOS want to lower the barrier to BCI research by providing all needed tools in one package. A detailed comparison is beyond scope, but notable differences: NeurOS puts a big emphasis on **foundation models and multi-modal data**, which as of the last info, MetaBCI might not have (MetaBCI focuses on EEG and standard algorithms, plus an experiment UI for common paradigms like P300, SSVEP). NeurOS also integrates things like Kafka, REST API, which MetaBCI likely doesn't. On the other hand, MetaBCI has a large collection of dataset loaders and may integrate more with MOABB (Mother of All BCI Benchmarks) for standardized evaluation. In code design, both are Pythonic; NeurOS might be more **async/real-time oriented**, whereas MetaBCI might run more in blocking mode for online experiments (this would need verification). The existence of MetaBCI shows there is a trend towards comprehensive BCI platforms – NeurOS is certainly in that conversation and would be evaluated against it. NeurOS's adoption of certain standards (NWB, BIDS) and its emphasis on new model types might give it an innovation edge. But MetaBCI already being cited in academic contexts means NeurOS will need to clearly demonstrate its improvements (e.g. better performance, more extensibility, or broader applicability).

In summary, **NeurOS compares very favorably to existing tools** in terms of features and modern design. It effectively combines capabilities that were previously scattered: MNE's signal processing, Braindecode's deep learning, BCI2000's real-time loop, plus novel additions like foundation models and multi-modal integration. The comparison table in NeurOS's docs succinctly illustrates this, showing NeurOS as the only one (among MNE, BCI2000, Braindecode) that ticks all the boxes for streaming, deep learning, multi-modality, transfer learning, CLI/API, etc. [47]. This breadth is arguably NeurOS's key strength. The flip side is that being a "Jack of all trades" can make it a large and complex project, whereas each existing tool is more narrowly optimized (MNE for analysis, BCI2000 for stability, etc.). If judging for a high-profile venue, one would commend NeurOS for advancing the state-of-the-art in BCI software integration, but also likely question how well each part performs versus dedicated tools (e.g., is NeurOS's EEG filtering as reliable as MNE's, are its deep learning components as well-validated as Braindecode's?). Those are areas for thorough validation as the project matures.

Strengths and Innovations

NeurOS-v1 demonstrates several strengths that make it stand out as a research and development platform:

- **Comprehensive End-to-End Solution:** NeurOS covers the full BCI pipeline in one framework – from acquiring raw signals to producing real-time predictions, plus evaluation and data logging. This all-in-one approach reduces friction in developing BCI applications and experiments. A user can do everything (data collection, processing, modeling, and even dashboard visualization) using a consistent API/CLI, which is a huge convenience and accelerates iteration.

- **Modern, Extensible Architecture:** The use of an asynchronous agent-based architecture is both cutting-edge and practical. It yields high performance (concurrent pipeline processing) and makes the system flexible to extension. Adding new algorithms or devices doesn't require modifying the core; instead, one can extend base classes and plug in the new component. This is evidenced by the large number of drivers and models already integrated without spaghetti code – thanks to clear abstraction layers.

- **Multi-Modal and Adaptive by Design:** NeurOS is built from the ground up to handle *multi-modal data* (numerous drivers running in parallel) and to adapt to changing conditions (quality monitoring, adaptive thresholds). This makes it suitable for next-generation BCI research which is moving beyond single EEG streams to richer data (e.g., combining EEG with eye tracking, or EEG with fNIRS for hybrid BCIs). The Constellation demo scenario (synchronizing 8+ different data sources) is something few platforms can do easily, giving NeurOS a unique capability for complex experiments [11].

- **Integration of Latest ML Advances:** By incorporating foundation models and deep learning techniques, NeurOS keeps researchers at the frontier. Features like zero-shot learning, cross-session transfer learning, and self-supervised neural embeddings (via models like Neuroformer, POYO, CEBRA) are cutting-edge in neurotech. NeurOS not only makes these accessible but unifies them with classical methods for fair comparison. This could spark new research, e.g. benchmarking foundation models vs. traditional classifiers under identical real-time conditions – something that would be of high interest at venues like NeurIPS.

- **User-Friendly Tools and Transparency:** The presence of a CLI, rich documentation (including a Quickstart and technical docs), and interactive dashboard indicates a focus on user experience. NeurOS tries to cater to both novices (who might prefer a no-coding interface or examples) and advanced users (who can dive into the Python API or integrate the server into custom applications). Moreover, because NeurOS is open-source (MIT-licensed) and self-contained, all algorithms are transparent – researchers can inspect and verify what's under the hood of each model or processing step, which is important for scientific trust. This transparency aligns with the norms of high-quality research software (reproducibility and openness).

- **Performance and Benchmarking Emphasis:** Real-time BCI requires efficiency, and NeurOS's design choices (async processing, queue backpressure, etc.) show careful consideration of performance. The built-in benchmarking suite is a strong asset: it not only helps developers optimize NeurOS itself, but also allows end users to measure how the system performs in their environment or against other solutions [37]. By facilitating objective performance metrics (latency, throughput, accuracy), NeurOS encourages a more rigorous evaluation of BCI pipelines, which is very much in spirit with the expectations of top-tier conferences.

Taken together, these strengths suggest that NeurOS-v1 is a **significant step forward** in BCI infrastructure. It does many things right that previous tools did not, or could not easily, and it clearly has been designed with both research **innovation** and real-world **applicability** in mind. A NeurIPS reviewer would likely laud the ambition and the execution of key ideas (like modular agents, multi-modality, new ML models) as it addresses longstanding needs in the field.

Limitations and Areas for Improvement

Despite its impressive scope, NeurOS-v1 has several areas that warrant improvement or cautious consideration:

- **Maturity and Reliability:** As a new, complex codebase, NeurOS is not yet as **battle-tested** as some existing tools. The test coverage is currently low (~40% of tests passing) and several features are marked as "in progress" or untested (e.g., the web API authentication, the multi-modal demo, some notebook generation functions) [12] [39]. This means bugs and stability issues likely remain. For deployment in critical applications (e.g. clinical use) or competition settings, more validation is needed. Increasing automated test coverage (unit and integration tests), running soak tests for long-term stability, and performing hardware-in-the-loop testing with actual devices will all be necessary to build confidence in the platform's robustness.

- **Documentation and Learning Curve:** While the project has a decent README and some docs, it currently lacks a comprehensive tutorial set, API reference, and contribution guide (though these are noted to be in development) [48]. Given the system's complexity, new users might struggle to utilize all features without more guidance. For example, setting up a multi-modal experiment with Kafka integration is non-trivial – a detailed runbook (which is partially provided in `docs/runbook_constellation.md`) is needed, along with simpler examples. Improving documentation (narrative tutorials, how-to guides for common use cases, and clarifying which parts are optional or how to install extras) is important for wider adoption. In an academic review, one might point out that the clarity of exposition (in a paper or documentation) needs to match the code's technical achievements, otherwise users may not realize or be able to reproduce the full capabilities.

- **Scope and Complexity:** NeurOS is almost *too comprehensive* in one package, which leads to a large dependency footprint and potential complexity in maintenance. Installing NeurOS with all features brings in Python packages for web servers, dashboard GUI, deep learning, hardware interfacing, etc. – this could pose environment conflicts or simply be burdensome if a user only needs a subset of the functionality. The "all-in-one" nature can also make the development team's job harder: every new advancement (be it a new model or a new device) has to be integrated and kept consistent with the whole. There is a risk of the project becoming unwieldy or **cumbersome** to manage as it grows. Users might experience parts of the system they don't need as "bloat." For instance, someone interested only in offline decoding might prefer not to install Kafka or FastAPI, but currently those are part of the full install (though some are optional extras). We discuss potential modularization in the next section as a remedy for this.

- **Performance Tuning and Constraints:** While the design is performance-aware, actual throughput/ latency will depend on the scenario. If very high channel counts or high-frequency data are used (say hundreds of channels EEG at 1kHz), pure Python might struggle unless careful optimizations (vectorization, possibly C/C++ extensions) are in place. Deep learning models can introduce non-deterministic latencies due to GPU scheduling, etc. NeurOS would benefit from performance benchmarking across various scenarios to identify bottlenecks. Perhaps it could incorporate more parallelism (e.g., using multiple processes or threads for certain CPU-bound tasks) if needed. As of now, it's unclear how it scales – for typical BCI (like 64-channel EEG at 250 Hz) it's fine, but pushing beyond that needs evaluation. An academic critique might ask for detailed latency/throughput results in the paper to be convinced of real-time guarantees under heavy loads.

- **Missing Features or Refinements:** There are a few things on the roadmap that highlight current gaps. For example, **model persistence** is only partly done – you can train models but not store them in a managed way (aside from pickling manually) [31]. A model registry or at least standardized save/ load commands would be important for practical use (so you don't have to retrain every time or can deploy the same model later). Data management could be improved: while raw data can be saved (NWB), the framework could offer dataset management utilities (maybe integration with MLflow or other experiment tracking, which might actually already be considered given references to database logging and metrics). Also, certain advanced BCI needs like adaptive feedback to the user (closing the loop with stimuli or neurofeedback) are not explicitly addressed in NeurOS's current feature set – one might need to extend it to cover those. These are not fundamental flaws, but rather areas where NeurOS could expand or polish its offering.

- **Community and Support:** Since NeurOS is new, it doesn't yet have a large user community. Tools like MNE or BCI2000 benefit from many users contributing fixes and improvements over years. NeurOS will need to cultivate a community to achieve long-term success – encouraging contributions (they plan a CONTRIBUTING.md) and ensuring the project is easy to build and test by others. The complexity of the project might intimidate new contributors unless modularized or clearly documented.

From a NeurIPS review standpoint, these limitations mostly boil down to *execution and scope concerns, not conceptual flaws*. The concept and design are strong, but the implementation needs further hardening and simplification in places. Reviewers might suggest focusing on ensuring reliability of key functionalities (so that the impressive list of features all demonstrably work as claimed) and maybe refining the scope to core strengths (to avoid trying to do everything at risk of spreading too thin). The next section addresses one of the key strategic questions: should NeurOS remain an integrated monolith or be split into a modular toolset to better manage complexity?

Monolithic vs. Modular Architecture Discussion

NeurOS-v1 is presented as a **single, large repository and package** that includes everything: drivers, processing, models, CLIs, servers, etc. This monolithic approach has clear advantages: it ensures that all components are tightly integrated and versions of each part are in sync. Users get a "batteries-included" experience – install one package and you have all the pieces working together. It also aids transparency and reproducibility: because all code (from data handling to model internals) is in one place, it's easy to inspect how something is implemented or to modify it for custom needs. In scientific software, having a one-stop toolkit can be very convenient (e.g., one namespace `neuros` where you can import everything). This integration likely made development easier initially too, because the maintainers could iterate on the pipeline holistically without worrying about cross-package compatibility.

However, as the project grows, a monolithic design can become **cumbersome**. NeurOS is already quite large in scope, and not every user will need every feature. For instance, a neuroscientist interested in offline decoding might want to use NeurOS's models and processing on stored data, but has no need for the real-time orchestrator or Kafka integration. Conversely, an engineer building a closed-loop system might only care about streaming and simple classifiers, and not use the deep learning models. With the current design, both users have to install and navigate the entire codebase. This raises the barrier in terms of system requirements (installing e.g. PyTorch, FastAPI, Streamlit, etc., even if not used). It also means the maintainers must ensure that changes in one part (say adding a new deep model) don't unexpectedly break another part (maybe the CLI or server). Monolithic codebases can become **fragile** as they expand, unless very rigorously tested and managed.

A logical solution is to adopt a more **modular (or plugin-based) architecture** moving forward. This doesn't necessarily mean abandoning the unified vision, but rather structuring the project into **separable components or packages** that can be developed and used somewhat independently. For example, NeurOS could be refactored into a few Python packages: - `neuros-core`: containing the core Pipeline class, orchestrator, base agents, base drivers (perhaps a couple of simple drivers like a FileDriver or MockDriver), and base models (maybe just a simple classifier). This would have minimal dependencies (numpy, asyncio, sklearn, etc.) and serve as the lightweight foundation. - `neuros-drivers` (or `neuros-hw`): containing all the hardware-specific drivers (BrainFlow EEG integration, LSL support, camera, etc.). This could depend on the core and bring in additional requirements like `brainflow`, `pylsl`, `opencv`, etc. Users who need hardware integration install this extra; others can skip it. - `neuros-models`: containing the more advanced models (deep learning models like EEGNet, Transformer, LSTM) and the foundation model wrappers (POYO, NDT, etc.). This would depend on core and bring in `torch` and any model-specific libs (like `torch_brain`). If someone only wants classical ML with no deep learning, they wouldn't need this package. - `neuros-ui`: containing the dashboard (Streamlit app code) and possibly the FastAPI server. This would depend on core and bring in `streamlit`, `fastapi`, etc. People running headless or just doing offline analysis could avoid installing UI components. - `neuros-utils` or others: possibly separate out things like data I/O (NWB, BIDS handling) into a submodule, or the auto-configuration system could be its own small piece.

These could still live in the same GitHub repository (as a monorepo with sub-packages) or be split into multiple repos under a Git organization. The key is they would be **separately installable** (and versioned). The maintainers could release them together or on different schedules if needed (though likely together for compatibility). From a user perspective, this means more flexibility: one could `pip install neuros-core` for a minimal setup, and then add `neuros-models` if deep learning is needed, etc., or just `pip install neuros[all]` to get everything. There is already a hint of this in the current setup (the README shows `pip install neuros[all]` as an option [49] and the audit recommends populating `extras_require` in setup.py [50]). So moving to a multi-package distribution is a natural extension of that idea.

The benefits of modularizing into a library of packages include: - **Reduced Footprint for Users:** They install only what they need, which can avoid heavy dependencies and potential version conflicts. For instance, someone not using the dashboard wouldn't need to pull in Streamlit and its dependencies. - **Easier Maintenance and Testing:** Smaller components mean the test suite for each can be run faster and in isolation. It's easier to pinpoint issues and manage contributions if the areas are more delineated. A bug in a driver won't hold up releasing a new version of the core pipeline, for example. - **Encouraging Contribution and Adoption:** External collaborators might contribute new drivers or models more readily if they see a clean separation (perhaps they only care about the `neuros-drivers` part). Also, other projects might use parts of NeurOS without having to adopt it wholesale. For example, an existing lab software might use `neuros-core` to get the orchestrator and then integrate its own analysis, without being forced into NeurOS's entire ecosystem. - **Modularity in Research**: From a scientific perspective, breaking it into components can encourage **modular experiments**. E.g., one could swap out the entire `neuros-models` module with a custom alternative if they want to test a radically different decoding approach, while still using `neuros-core` for streaming. It's analogous to how in ML, you have separate libraries for core computations vs. model zoos vs. application frameworks.

Of course, there are also reasons to keep things integrated. A single repo/package ensures that all parts are compatible and versioned together, which avoids dependency hell for the user. If split too much, there's overhead in making sure the versions of core/models/etc. all line up. Also, discoverability can be an issue: users might not realize they need to install an extra for some feature (this can be mitigated by good documentation and perhaps a meta-package that installs all). Given NeurOS's target audience (researchers who may not be software engineers), a one-liner installation is attractive.

A **middle-ground** approach could be to maintain a single code repository (for coherence and easier cross-component changes) but structure it such that it produces multiple installable distributions. Many large projects do this (for example, Apache Airflow has many optional components as separate pip packages, but developed in one repo). NeurOS could continue as one repo under, say, a "NeurOS" organization on GitHub, with directories for each sub-package. Each sub-package could even have its own README for clarity. The main README could then explain the various install options and combinations. This way, the project remains unified in vision but modular in usage.

To directly answer the question: *Does it make sense to split NeurOS into multiple components or keep it integrated?* – **It likely makes sense to modularize it into a suite of packages** as the project matures. The current monolithic design made sense to quickly develop a cohesive platform, but as features grow, modular design will improve manageability and user experience. Many successful toolkits follow this trajectory: they start monolithic for initial development and integration, then refactor into a more plugin-oriented architecture. NeurOS is reaching the point where splitting out packages (especially heavy or independent ones like hardware support and advanced models) would be beneficial. This doesn't diminish the integrated user experience – indeed, if done right, the user might hardly notice except that installation is lighter and the structure clearer.

Suggestions for creating a library of packages: 1. Identify logical groupings (core vs. extras as outlined above). Ensure the core has stable APIs that the extras can depend on. For instance, finalize the `BaseDriver` and `BaseModel` interfaces in core, so external packages can reliably implement against them. 2. Use Python's `entry_points` or plugin mechanisms for discovery if needed (e.g., the core orchestrator could dynamically detect additional drivers or models if they register themselves, so that plugins "feel" native). 3. Adjust the build/CI process to test each component separately and together. This will guard against integration issues. 4. Provide clear documentation on how to install and what each optional module contains, perhaps with real-world scenarios (e.g., "if you want to use NeurOS with an OpenBCI headset, install neuros-core and neuros-drivers; if you also want deep learning models, install neuros-models"). 5. Maintain a meta-package or a default installation that includes the most commonly used pieces, so newcomers get a good experience (for example, `pip install neuros` might include core + drivers + a couple models by default, reserving the really heavy ones for an `[all]` or separate pip install).

By moving in this direction, NeurOS can remain **integrated in vision but modular in implementation**, which is likely the ideal balance. This will make the tool less "large and cumbersome" as the user query put it, without losing the benefit of having everything work well together.

Conclusion

NeurOS-v1 is an ambitious and forward-thinking BCI platform that **marries real-time data acquisition with state-of-the-art machine learning in a unified system**. From a high-level perspective, it brings significant improvements to the BCI software landscape: a cleaner architecture, support for multiple data modalities, integration of both classical and novel decoding methods, and attention to performance and usability. In many ways, NeurOS is designed as a *researcher's toolbox* for the coming era of BCIs that leverage complex models and diverse data streams – a vision very much aligned with current trends highlighted at venues like NeurIPS.

In evaluating its design and implementation, we found that NeurOS excels in **modularity, extensibility, and feature-richness**. It has a strong core architecture (agent-based, asynchronous) that is well-implemented, and it smartly extends that core with numerous plugins (drivers, models, etc.) to cover a wide range of use cases (from simple EEG classification to multi-modal sensor fusion). The platform compares favorably against both older and contemporary tools, often exceeding them in functionality (for instance, offering real-time streaming plus deep learning capabilities that others lack [15]). The code quality appears high, with modern Python practices and good documentation of internals, which is crucial for a project of this scope.

On the flip side, NeurOS is still **in development** and certain aspects (testing, documentation, deployment guides) need to catch up to the core features. To truly shine as a production-ready or widely-used research tool, the team will need to polish these facets, ensuring the impressive list of features is matched by stability and ease of use. Splitting the project into a set of coherent Python packages could be a wise strategy to manage its growing complexity and cater to different user needs without overload. This modular approach would likely make NeurOS more sustainable and invite broader collaboration.

In conclusion, as if delivering a verdict for a top conference: **NeurOS-v1 is a highly promising platform that substantially advances the state-of-the-art in BCI software infrastructure**. It effectively demonstrates how to integrate heterogeneous data streams with advanced neural network models in real time, all within an elegant unified framework – an achievement that addresses a clear gap in the BCI field. The work is technically sound and practically relevant, though not without areas to refine. With continued development to improve its robustness and perhaps a restructuring into modular components for scalability, NeurOS has the potential to become a cornerstone toolkit for both BCI researchers and engineers. It embodies a much-needed modernization of BCI "operating systems," and with the suggested improvements, it could set a new standard for how we design and deploy brain-computer interface pipelines in research and beyond.

Sources: The analysis above is based on the NeurOS-v1 repository documentation and code, including the project README [1] [11], official docs [47] [26], and a recent audit report [2] [48] which detail the system's features, comparisons, and development status.

---

[1] [11] [12] [14] [40] README.md
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/README.md

[2] [7] [8] [9] [13] [24] [31] [36] [37] [38] [39] [41] [43] [44] [45] [48] [50] AUDIT.md
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/AUDIT.md

[3] [4] [10] [19] [21] [22] [33] [34] [35] orchestrator_agent.py
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/neuros/agents/orchestrator_agent.py

[5] [6] [18] base_driver.py
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/neuros/drivers/base_driver.py

[15] [20] [26] [27] [30] [32] [42] [47] [49] index.md
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/docs/index.md

[16] [17] [25] [28] poyo_model.py
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/neuros/foundation_models/poyo_model.py

[23] pipeline.py
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/neuros/pipeline.py

[29] chatgpt-eval.txt
https://github.com/sidhulyalkar/neurOS-v1/blob/d3b833ace7c60620624f49fce3a1b673e275deba/chatgpt-eval.txt

[46] GitHub - TBC-TJU/MetaBCI: MetaBCI: China's first open-source platform for non-invasive brain computer interface. The project of MetaBCI is led by Prof. Minpeng Xu from Tianjin University, China.
https://github.com/TBC-TJU/MetaBCI