AUTOBCI: FORECAST-GUIDED AGENTIC NEURAL ARCHITECTURE DISCOVERY FOR EEG-BASED BRAIN–COMPUTER INTERFACES

Anonymous authors
Paper under double-blind review

ABSTRACT

EEG-based brain–computer interfaces support a broad range of applications, yet designing decoding architectures that perform well across diverse tasks remains challenging. We introduce AutoBCI, an agentic framework in which a Designer Agent and a Forecaster Agent support the discovery and selection of EEG decoding architectures across tasks. The Designer Agent performs Pool-Guided Architecture Discovery (PGAD), generating and refining architectures through training and validation across multiple EEG tasks, such as emotion recognition, motor imagery, and sleep staging. The Forecaster Agent performs Performance Estimation from Early Knowledge (PEEK), using architecture code, the training protocol, and early learning curves to predict full-budget validation performance and select promising candidates for continued training. Across 14 EEG datasets spanning motor imagery, emotion recognition, and sleep staging, we evaluate AutoBCI with six LLMs, including Opus 5.5 and GPT 5.6 Sol, and compare the architectures selected by the search procedure against ten baselines: six conventional EEG models and four foundation models. The architecture discovered by AutoBCI with Claude Opus 5.5 achieves 64.16% average test balanced accuracy (bAcc), compared with 63.87% for REVE, the strongest baseline on this metric. Using ten observed epochs, PEEK reduces mean absolute error in predicting average validation bAcc from 2.20 to 1.36 percentage points, a 38.1% reduction relative to the best-observed-score baseline.

1 INTRODUCTION

Electroencephalography (EEG) is a key modality for non-invasive brain–computer interfaces. EEG decoding spans tasks with different spatial coverage, temporal scales, and label spaces. For over a decade, researchers have encoded domain knowledge through selected frequency bands, temporal windows, and spatial filters, such as common spatial patterns for motor-imagery decoding (Schirrmeister et al., 2017). Deep learning learns features directly, but model structures still reflect human design: EEGNet uses compact spatiotemporal convolutions (Lawhern et al., 2018), TSception captures multiple temporal scales and spatial asymmetry (Ding et al., 2022), and DeepSleepNet models sleep-stage dependencies with convolutional and recurrent layers (Supratak et al., 2017). These advances demonstrate **the value of EEG-specific architectural design**, while **selecting suitable structures across tasks remains a manual burden**. This motivates automated discovery of architectures that perform well across different EEG tasks.

Agentic AI offers a way to reduce the manual effort involved in working across these settings. Existing systems demonstrate that **agents can automate EEG analysis workflows**. EEGAgent supports signal exploration, event detection, and report generation (Zhao et al., 2026); CogEEGAgent translates user questions into registered analyses with independent confirmation (Hou et al., 2026); and NS-Copilot coordinates specialized agents and pretrained models for neuroscience workflows (Liu et al., 2026). These capabilities improve access to analysis tools, but **these systems focus on automating analysis rather than designing decoding architectures**. Reducing the model-design burden requires extending this automation to the architectures themselves, using experimental feedback to guide structural changes.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Work on neural architecture search takes this further, showing that **architecture design can be automated through search and feedback**. CTNAS-EEG introduces a compatible search space and constrained search procedure across EEG tasks (Duan et al., 2023). NeuroWeaver evolves executable EEG pipelines using domain-informed initialization and performance, novelty, and efficiency objectives (Wang et al., 2026). More broadly, EvoPrompting evolves architecture code, while NADER uses collaborating agents to guide architectural modifications (Chen et al., 2023; Yang et al., 2025). However, **success on separately optimized tasks does not establish the quality of a shared architecture**. Our goal is to discover a single architecture that performs well across EEG tasks. We therefore train and validate each candidate separately for each task, then use the combined validation results to guide architecture selection and refinement. In this study, we evaluate this approach on three common EEG tasks: emotion recognition, motor imagery, and sleep staging. Evaluating each proposal across these tasks increases the training cost of discovery, making candidate screening an important part of the search.

To reduce this evaluation cost, prior work exploits the finding that **partial training evidence can support early performance prediction**. Learning-curve extrapolation identifies unpromising runs (Domhan et al., 2015), and performance predictors combine architecture features, hyperparameters, and partial validation trajectories (Baker et al., 2017). LLM-ENAS-PKB further uses language models as architecture-performance surrogates (Weilin et al., 2026). However, **forecast accuracy alone does not establish reliable candidate selection**. For screening to be useful, promising candidates must remain in the search. We therefore use architecture code and early learning curves to predict how well a candidate will perform after full training. These forecasts help identify promising candidates for continued training. We evaluate both prediction accuracy and whether the selected candidates retain the architectures that perform best after full training.

To address these challenges, we introduce **AutoBCI**, an agentic framework containing two agents: a Designer Agent that uses Pool-Guided Architecture Discovery (PGAD) to generate and refine EEG architectures, and a Forecaster Agent that uses Performance Estimation from Early Knowledge (PEEK) to predict candidate performance and guide continued training, as shown in Figure 1. First, **PGAD automates architecture design through executable code generation and refinement**. The agent proposes models and uses measured validation feedback to revise their structure, extending its role from analysis workflow execution to model design. Second, **PGAD selects shared architectures using joint evidence across tasks**. Each candidate is trained and validated independently on emotion recognition, motor imagery, and sleep staging. An equal-task validation objective ranks a persistent candidate pool, from which strong parents are refined alongside fresh proposals. Third, **PEEK guides training allocation using early performance forecasts**. It combines architecture code, the training protocol, and early learning curves to predict full-budget validation scores and rank candidates for continued training. These mechanisms connect model generation, cross-task evaluation, and candidate screening within one discovery framework.

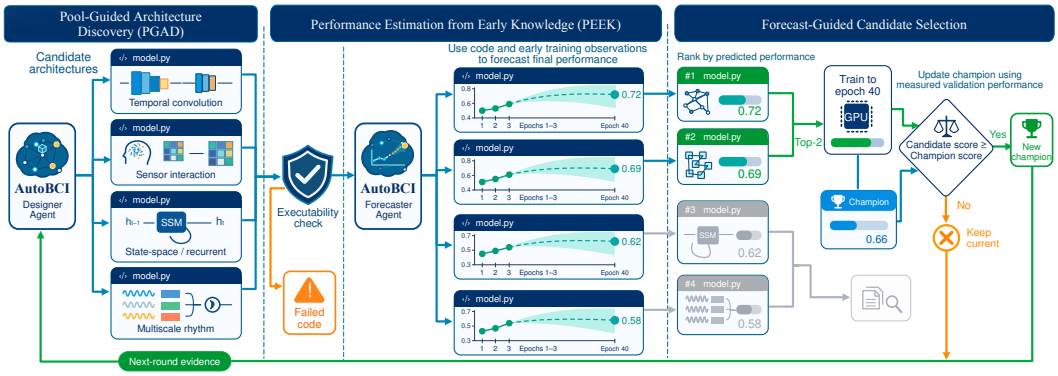


Figure 1: **AutoBCI framework overview.** The Designer Agent uses PGAD to generate and refine EEG architectures; the Forecaster Agent uses PEEK to predict performance and guide continued training. Validation feedback guides subsequent discovery rounds.

Our contributions are threefold. First, we introduce AutoBCI and develop PGAD for agentic EEG architecture discovery, using a persistent candidate pool and cross-task validation feedback to guide

code generation and refinement. Second, we develop PEEK to forecast full-budget performance from architecture code and early learning curves, enabling candidate screening before training is complete. Third, we evaluate six LLM-guided searches across 14 datasets and compare their champions with ten baselines. The searches produce 269 architectures trained on all three task families; the strongest champion achieves 64.16% average test bAcc, exceeding the strongest baseline by 0.29 percentage points. PEEK reduces forecast mean absolute error by 38.1% after ten epochs. Retaining three candidates per round preserves a full-budget winner in 91.7% of rounds, with an estimated 44.9% reduction in training epochs.

2 METHOD

AutoBCI combines a Designer Agent for architecture discovery with a Forecaster Agent for early candidate selection. The Designer Agent uses PGAD to generate architectures, evaluate separately trained instances across EEG tasks, and refine candidates using their combined validation scores. The Forecaster Agent uses PEEK to predict full-budget validation performance from architecture code, the training protocol, and early learning curves, helping select candidates for continued training.

2.1 POOL-GUIDED ARCHITECTURE DISCOVERY

PGAD organizes architecture discovery around a persistent pool of evaluated candidates. Starting from an initial set of proposals, the Designer Agent uses validation feedback across tasks to identify strong architectures and propose refinements. Each subsequent round combines these refinements with fresh designs, allowing the search to build on previous discoveries while exploring alternatives. The procedure consists of candidate generation, cross-task evaluation, and pool-based selection and refinement, as detailed below.

Candidate generation. PGAD runs for R discovery rounds, indexed by $r \in \{1, \dots, R\}$, with N candidate architectures proposed per round. Round $r = 1$ contains N fresh proposals from the configured LLM. Each proposal contains complete PyTorch model code and a short architectural hypothesis. The model interface accepts configurable channel, sample, and class counts, mapping a batch of EEG signals to class logits. Before training, the framework checks the response format and model interface. Candidate code is preserved as generated; implementation errors are recorded as candidate failures.

Evaluation across tasks. For each candidate, the same architecture source is trained independently on K task families, indexed by $k \in \{1, \dots, K\}$, with separate model parameters and checkpoints for each task. Our evaluation uses emotion recognition, motor imagery, and sleep staging ($K = 3$). Let $\mathcal{D}_k$ denote the datasets in task family k, E the full training budget in epochs, and $b_{a,d}(e)$ denote the validation balanced accuracy of architecture a on dataset d at epoch e. At each epoch, datasets within a task receive equal weight. The task score is the highest such mean over the training run:

$$s_{a,k} = \max_{1 \le e \le E} \frac{1}{|\mathcal{D}_k|} \sum_{d \in \mathcal{D}_k} b_{a,d}(e). \tag{1}$$

Thus, all datasets within a task share one selected checkpoint epoch, while different tasks may select different epochs. The architecture score assigns equal weight to the K task scores:

$$S_a = \frac{1}{K} \sum_{k=1}^{K} s_{a,k}. \tag{2}$$

An architecture is eligible for selection only after its training runs on all K tasks complete E epochs.

Selection and refinement. Before round $r > 1$, eligible architectures from all preceding rounds form a shared pool $\mathcal{P}_{r-1}$ of size $M_{r-1} = |\mathcal{P}_{r-1}| \le N(r-1)$, as illustrated in Figure 2. The round selects the P highest-scoring architectures by S_a as parents and requests Q refinements per parent. Each refinement receives the parent's source code, learning curves, and task-specific validation results, and is prompted to change one architectural component and describe the intended change. Alongside these PQ refinements, the LLM generates U fresh explorations, with

$$N = PQ + U. \tag{3}$$

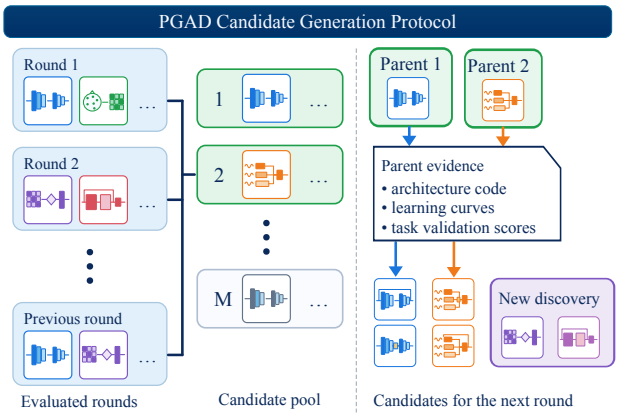


Figure 2: **PGAD candidate generation protocol. Left:** Architectures evaluated in previous discovery rounds. **Middle:** A shared candidate pool ranked by combined validation performance. **Right:** Top-ranked parents provide source code, learning curves, and task-specific validation scores for targeted refinements. These refinements are combined with fresh architecture proposals to form the next candidate batch.

Thus, the search requests RN architectures over R rounds. The highest-scoring eligible architecture in $\mathcal{P}_R$ is the final champion. Validation data determine parent and champion selection; the held-out test split is used only for a separate final evaluation.

2.2 Performance Estimation from Early Knowledge (PEEK)

PEEK uses early performance forecasts to select promising architectures for continued training. After a short initial training period, the Forecaster Agent combines each candidate's architecture code, training protocol, and observed learning curves to estimate its full-budget validation performance. Architecture code describes the model's design, while the curves show how that design learns on the target task. PEEK uses this complementary evidence to rank candidates and allocate the remaining training budget to the predicted leaders. The procedure has two steps: performance forecasting and candidate selection.

Performance forecasting. For task k, each candidate a first trains for $e_0 < E$ epochs. Let C_a denote its source code and $\mathcal{H}_{a,k}^{1:e_0}$ its observed training and validation history. Given the training protocol π_k, PEEK produces

$$\widehat{s}_{a,k} = F_{\text{LLM}}\left(C_a, \mathcal{H}_{a,k}^{1:e_0}, \pi_k\right), \tag{4}$$

where the target $s_{a,k}$ is the best equal-dataset validation balanced accuracy at a common checkpoint within E epochs (Equation (1)). Aligning the forecast with this selection criterion lets PEEK assess a candidate's potential over the remaining budget, including improvement beyond its early score. Predictions lie in $[0, 1]$ and must be at least the best validation score already observed. The LLM receives only candidate code, the specified protocol, and the observed portion of the learning history. For shared-architecture selection, we average the task forecasts using the same equal-task weighting as Equation (2):

$$\widehat{S}_a = \frac{1}{K}\sum_{k=1}^{K} \widehat{s}_{a,k}. \tag{5}$$

Candidate selection. To select shared architectures, PEEK ranks candidates by the combined forecast $\widehat{S}_a$ (Equation (5)). The screening policy retains the N_{keep} highest-ranked candidates and continues each on all K tasks to E epochs, where $1 \le N_{\text{keep}} \le N$. Only candidates that complete all K tasks enter the PGAD pool, using their measured scores S_a; partially trained candidates remain ineligible for parent and champion selection. For N candidates initially trained for e_0 epochs on each task, the nominal cost is $K[Ne_0 + N_{\text{keep}}(E - e_0)]$ task-epochs, compared with KNE for full training. The retention–cost analysis sweeps $N_{\text{keep}} = 1, \ldots, 6$, with the abstract reporting the three-candidate operating point. Separate task-level diagnostics rank candidates by $\widehat{s}_{a,k}$ and measure top-one and top-two retention for each task.

Table 1: EEG datasets, task-specific inputs, and class counts.

Task	No.	Datasets	Unified input	Classes
Motor Imagery	1	BCIC-IV-2a (Tangermann et al., 2012)	65 ch × 4 s	Left, Right, Foot, Tongue, Cylin, Sphe, Lumbrical (7 classes)
	2	OpenBMI-MI (Lee et al., 2019)		
	3	BCIC-Upperlimb (Jeong et al., 2022)		
	4	Cho2017 (Cho et al., 2017)		
	5	HighGamma (Schirrmeister et al., 2017)		
	6	PhysioNet-MI (Schalk et al., 2004)		
	7	SHU-MI (Ma et al., 2022)		
	8	Shin2017A (Shin et al., 2016)		
Emotion Recognition	9	SEED (Duan et al., 2013)	65 ch × 4 s	Positive, Neutral, Negative, Sad, Fear, Happy, Disgust (7 classes)
	10	SEED-IV (Zheng et al., 2018)		
	11	SEED-V (Liu et al., 2021)		
Sleep Staging	12	Sleep-EDF (Kemp et al., 2000)	8 ch × 30 s	Wake, N1, N2, N3, REM (5 classes)
	13	ISRUC (Khalighi et al., 2016)		
	14	HMC (Alvarez-Estevez & Rijsman, 2021)		

3 Experimental Setup

Our experiments evaluate architecture discovery and early candidate selection on shared EEG datasets. PGAD evaluates candidates with a 40-epoch budget. Completed runs establish the full-budget scores and winners needed to evaluate PEEK. The forecaster receives only architecture code, the training protocol, and early learning curves; full-run outcomes provide ground truth for retrospective evaluation of forecast accuracy and winner retention.

3.1 Dataset construction

EEG tasks differ in their prediction targets and recording characteristics, so we organize the 14 datasets into motor imagery (MI), emotion recognition (ER), and sleep staging (SS), as shown in Table 1. Datasets within each family are pooled to train one model instance per task. Channel layouts and labels are harmonized within a task. MI and ER recordings are mapped to the 65-channel standard 10–10 system template used by LEAF (Jiang et al., 2025), while the sparser sleep recordings use an eight-channel template. Validation and test scores are reported separately for each dataset.

All candidates use the same fixed training, validation, and test partitions and sleep subsets. Training fits model parameters, validation selects checkpoints and architectures, and held-out test data are used only for final evaluation. Preprocessing and split construction are detailed in Supplementary A.5.

3.2 Architecture search protocol

On these datasets, PGAD searches share a common budget to support comparison of architecture discovery across LLMs. Each search comprises $R = 6$ discovery rounds with $N = 8$ candidates per round, evaluated across $K = 3$ task families (MI, ER, and SS). This gives $RN = 48$ architecture proposals and up to $KRN = 144$ independent task-training runs. Round 1 contains $N = 8$ fresh proposals. For later rounds, $P = 2$ parents each receive $Q = 3$ refinements, and $U = 2$ fresh proposals complete the batch ($N = PQ + U = 8$). The parameter constraint is strictly fewer than one million trainable parameters in every task-specific instantiation. Recurrent architectures are excluded, and each candidate is trained from scratch under the same task-specific protocol.

Each search uses one of the six language models listed in Table 2. The initial proposals are guided by the task names and class categories, input-shape constraints, training protocol, and architectural constraints. In subsequent rounds, the two selected parents provide architecture code and validation learning curves for all three tasks, allowing refinements to address their observed strengths and weaknesses. Fresh proposals preserve an opportunity to explore architectures beyond the selected parents. Candidates undergo validity and training-compatibility checks before training. Only architectures that successfully complete all three tasks are eligible for parent and final champion selection. Implementation interfaces, generation settings, and detailed checks are provided in Supplementary A.6; complete prompts appear in Supplementary A.10.

The six LLMs run architecture searches through their respective command-line interfaces, as shown in Table 2.

Table 2: LLMs and execution settings for architecture discovery.

Model tier	Provider	LLM	Agent interface	Input (USD/1M)	Output (USD/1M)
Flash	Anthropic	Claude Sonnet 5	Claude Code	2.00	10.00
	DeepSeek	DeepSeek V4.1 Flash	Claude Code	0.30	1.20
	Google	Gemini 3.8 Flash	Gemini CLI	0.75	3.75
	Alibaba	Qwen3.8 Flash	Claude Code	0.15	0.47
Flagship	OpenAI	GPT-5.6 Sol	Codex CLI	4.00	20.00
	Anthropic	Claude Opus 5.5	Claude Code	4.00	20.00

We use the default thinking levels for all models. DeepSeek and Qwen were accessed through Anthropic-compatible endpoints via Claude Code.

3.3 Training and evaluation protocol

To compare the architectures generated during search, all candidates follow a common training and evaluation protocol. Each candidate is trained independently on the three task families for 40 epochs using AdamW, a batch size of 512, and full BF16 precision. Validation performance selects the checkpoint for each task. Experiments run in PyTorch on NVIDIA RTX PRO 6000 Blackwell GPUs, with independent candidate/task jobs executed in parallel.

Performance is measured by balanced accuracy (bAcc), the mean recall over classes present in each split. Validation bAcc selects task checkpoints with equal dataset weights and architectures with equal task weights. Selected architectures are tested using these checkpoints, reporting bAcc, class-support-weighted F1 (wF1), and Cohen's Kappa with the same averaging; test data remain excluded from search and selection. Prompts, code, parent lineage, splits, configurations, seeds, learning curves, checkpoints, and outcomes are retained for reproducibility; further training and execution details appear in Supplementary A.7.

3.4 Baseline models and comparison settings

To assess the discovered architectures against established EEG decoders, we compare with ten baselines. The six conventional models are EEGNet (Lawhern et al., 2018), DeepConvNet (Schirrmeister et al., 2017), TSception (Ding et al., 2022), EEG-Conformer (Song et al., 2022), AttnSleep (Eldele et al., 2021), and FAST (Jiang et al., 2026). The four foundation models are LaBraM (Jiang et al., 2024), CBraMod (Wang et al., 2025), LEAF (Jiang et al., 2025), and REVE (El Ouahidi et al., 2026). Conventional models are trained from scratch, while foundation models are fine-tuned from pretrained checkpoints. All baselines use the same task partitions and validation-based checkpoint selection as AutoBCI. Training budgets, optimization settings, and input adaptations are detailed in Supplementary A.4.

4 Results

We evaluate the discovered architectures on held-out test data and analyze architecture discovery and early performance forecasting using validation data.

4.1 Performance of discovered architectures

To assess full-budget search, we compare validation-selected champions on held-out tests, weighting the three tasks equally; baseline results average three seeds. Opus achieves the highest average bAcc (64.16%), exceeding REVE by 0.29 percentage points (pp), while REVE leads on average wF1 and Cohen's Kappa, as shown in Table 3. Task leaders differ: Opus leads MI, REVE leads emotion, and AttnSleep has the highest sleep bAcc. Discovered champions span 57.69–64.16% average bAcc; in these searches, Gemini and DeepSeek achieve higher average test bAcc than GPT.

Table 3: **Performance comparison of small-model baselines, foundation models, and AutoBCI-discovered architectures.** Results are evaluated on held-out test sets for motor imagery, emotion recognition, and sleep staging. bAcc and wF1 are percentages. Per column: **first**, **second**, third.

Model	Spec	Motor Imagery			Emotion			Sleep			Average		
		bAcc	wF1	Kappa	bAcc	wF1	Kappa	bAcc	wF1	Kappa	bAcc	wF1	Kappa
EEGNet (Lawhern et al., 2018)	8.8K	57.83	57.12	0.233	37.97	34.48	0.164	69.09	70.60	0.640	54.96	54.07	0.345
TSception (Ding et al., 2022)	17.5K	54.58	53.63	0.167	43.93	45.91	0.256	66.75	70.74	0.648	55.09	56.76	0.357
DCN (Schirrmeister et al., 2017)	317K	66.70	66.36	0.396	42.73	38.03	0.215	**73.89**	76.68	0.697	61.10	60.35	0.436
FAST (Jiang et al., 2026)	475K	58.59	58.31	0.243	35.39	36.94	0.137	72.26	74.99	0.686	55.41	56.75	0.355
EEG-Conformer (Song et al., 2022)	2.13M	66.32	66.21	0.396	48.08	48.92	0.302	72.34	76.07	0.701	62.25	63.73	0.466
AttnSleep (Eldele et al., 2021)	2.32M	63.73	63.41	0.344	40.28	40.79	0.201	**74.24**	76.89	0.695	59.42	60.36	0.413
CBraMod (Wang et al., 2025)	4.89M	46.18	43.66	0.028	40.59	40.40	0.201	69.02	72.65	0.657	51.93	52.24	0.295
LaBraM (Jiang et al., 2024)	5.82M	47.68	44.38	0.053	49.24	49.85	0.323	70.80	75.24	0.677	55.91	56.49	0.351
LEAF (Jiang et al., 2025)	27.41M	67.17	67.05	0.408	49.13	**51.19**	0.327	72.57	75.85	0.687	62.96	64.70	0.474
REVE (El Ouahidi et al., 2026)	69.19M	**67.80**	**67.49**	**0.424**	**51.15**	**51.87**	**0.345**	72.67	**77.99**	**0.717**	**63.87**	**65.78**	**0.495**
AutoBCI (Claude Sonnet 5)	Flash	59.13	58.64	0.257	47.48	47.84	0.292	70.96	75.47	0.683	59.19	60.65	0.411
AutoBCI (DeepSeek V4.1 Flash)	Flash	61.74	61.44	0.304	47.62	48.72	0.296	72.09	76.12	0.696	60.49	62.09	0.432
AutoBCI (Gemini 3.8 Flash)	Flash	65.31	65.04	0.371	45.74	44.76	0.259	73.80	77.16	0.701	61.62	62.32	0.444
AutoBCI (Qwen3.8 Flash)	Flash	58.39	57.28	0.242	43.00	44.26	0.237	71.69	76.08	0.691	57.69	59.21	0.390
AutoBCI (GPT-5.6 Sol)	Flagship	63.69	63.48	0.337	44.25	44.76	0.244	73.11	76.19	0.689	60.35	61.48	0.423
AutoBCI (Claude Opus 5.5)	Flagship	**68.55**	**68.38**	**0.435**	**50.10**	51.06	**0.329**	73.83	**77.55**	**0.712**	**64.16**	**65.67**	**0.492**

4.2 Architecture discovery across language models

To trace architecture discovery, we examine six search rounds in Figure 3. Opus finishes highest at 66.26% combined validation bAcc (+2.98 pp), followed by Gemini (63.59%) and DeepSeek (63.10%); Qwen improves most (+16.90 pp) but finishes lowest (58.43%). Opus's round median rises from 61.62% to 65.47%, whereas Sonnet's fluctuates despite an improving best score. Both refinement and fresh exploration yield champions: Opus refines multiscale convolutions with channel gating and mean–standard-deviation pooling, while Gemini's fresh design combines filter banks, dilated residual blocks, and attention pooling. DeepSeek uses multiscale filters with covariance pooling. Model definitions and round-by-round identities appear in Supplementary A.9 and Supplementary Figure 6.

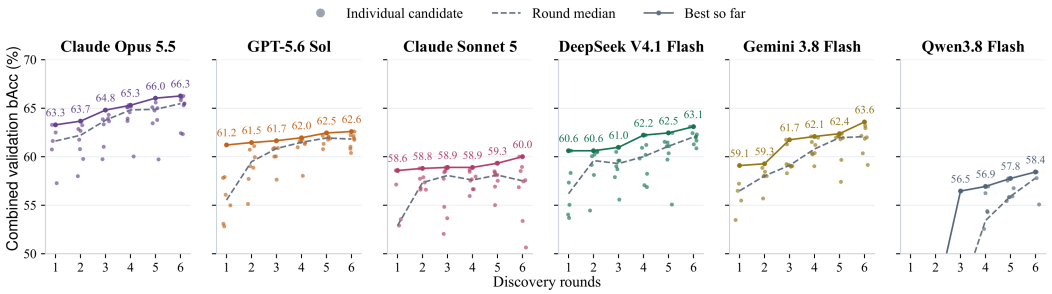


Figure 3: **Discovery trajectories across six LLM-guided searches.** Dots show candidates completed on all three tasks; dashed lines mark round medians, and solid lines track the best combined validation bAcc so far. Labels give the incumbent scores. The vertical axis is truncated at 50%; round-by-round architecture identities appear in Supplementary Figure 6.

Preserving progress and refinement paths. To examine the pool's role, we trace incumbent scores and parent reuse across all six searches in Table 4. DeepSeek's round-2 and Sonnet's round-4 best scores fall below their prior incumbents by 0.030 and 0.221 pp; the pool preserves the stronger candidates. All six searches reuse parents older than the preceding round. Opus, GPT, and DeepSeek champions follow rounds $1 \to 3 \to 4 \to 5 \to 6$, while Sonnet's follows $1 \to 2 \to 5 \to 6$. These refinement paths include parents selected from earlier rounds. Gemini and Qwen instead discover

fresh round-6 champions, showing how historical retention supports continued refinement alongside fresh exploration.

Table 4: **Historical retention and parent reuse in recorded searches.** Declines compare a round's best combined validation bAcc with the prior incumbent. Older-parent rounds use at least one parent from before the immediately preceding round. Arrows identify parent-to-child generation rounds in the champion's ancestry.

Model	Round-best declines	Older-parent rounds	Older-parent link in champion lineage
Claude Opus 5.5	None	R3	**Yes** ($1 \rightarrow 3$)
Claude Sonnet 5	R4 (0.221 pp)	R3, R4, R5, R6	**Yes** ($2 \rightarrow 5$)
DeepSeek V4.1 Flash	R2 (0.030 pp)	R3, R4, R6	**Yes** ($1 \rightarrow 3$)
Gemini 3.8 Flash	None	R3	None (fresh)
GPT-5.6 Sol	None	R3	**Yes** ($1 \rightarrow 3$)
Qwen3.8 Flash	None	R5	None (fresh)

4.3 Early performance forecasting and candidate selection

To evaluate early screening, we compare PEEK and the best-observed-score baseline against full-budget validation scores on paired successful rounds. Each LLM initially forecasts its own candidates from architecture code, the training protocol, and the observed learning-curve prefixes. After ten epochs, PEEK reduces combined-score mean absolute error (MAE) from 2.20 to 1.36 pp across 36 rounds (38.1%), anticipating gains beyond the observed scores, as shown in Figure 4A–B. The baseline uses each task's highest equal-dataset mean validation bAcc within the observed epochs as its full-budget prediction, then averages these predictions equally across the three tasks.

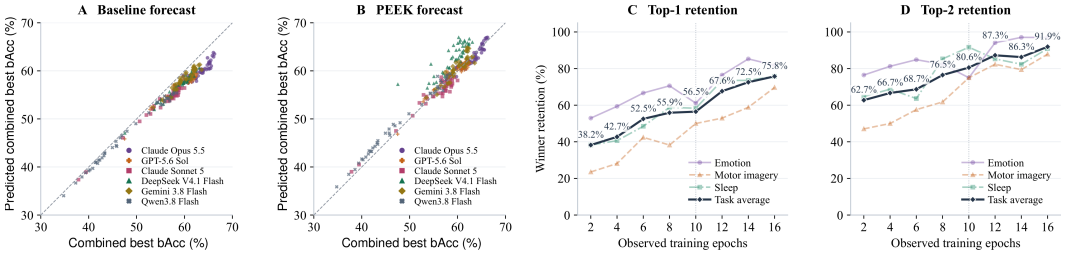


Figure 4: **Early forecasts and candidate retention.** **(A–B)** Ten-epoch forecasts versus actual combined validation bAcc for 269 architectures, averaging task-wise best-within-40 scores. **(C–D)** Top-one and top-two task-winner retention across observation windows; dark curves average the three tasks equally. Dotted lines mark ten epochs.

Forecasting gains vary across LLMs and observation windows, as shown in Figure 5. At ten epochs, PEEK lowers MAE for five of six searches, including Opus (3.27 to 0.79 pp) and GPT (2.52 to 0.67 pp), but increases it for DeepSeek (1.97 to 3.63 pp). By epoch 16, the baseline has lower aggregate MAE (0.88 versus 1.18 pp; 33 paired rounds), although PEEK remains better for Opus, GPT, and Sonnet.

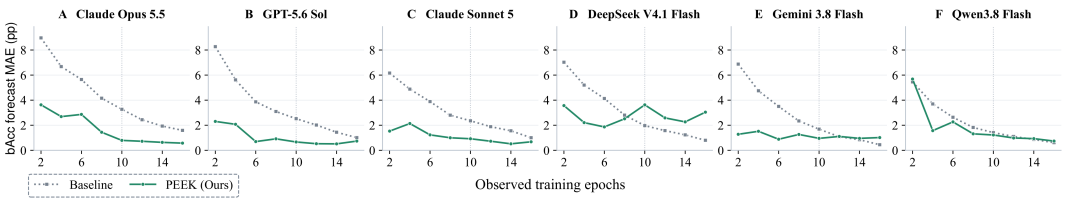


Figure 5: **Forecast error across LLMs and observation windows.** Combined validation bAcc MAE (pp), averaged over candidates within rounds and equally over paired rounds. Cohorts may vary by window; dotted lines mark ten epochs.

To assess selection quality, we rank candidates within each task by $\widehat{s}_{a,k}$. At ten epochs, retaining two rather than one increases task-average winner retention from 56.5% to 80.6%, as shown in Figure 4C–D; top-two retention is 75.0% for MI and emotion and 91.7% for sleep. At 16 epochs, top-one and top-two retention reach 75.8% and 91.9% across 33 paired rounds. The next subsection evaluates shared-architecture selection using combined scores $\widehat{S}_a$.

Separating the Designer and Forecaster. To test selection across LLMs, DeepSeek, GPT, and Gemini forecast 91 Opus and Sonnet candidates across 12 rounds using identical code and ten-epoch inputs to the self-forecasts. Each external forecaster achieves 72.2% task-average top-two retention, versus 75.0% for self-forecasting; top-one retention is 47.2–50.0% versus 50.0%, as shown in Table 5. External forecasts match or improve top-two retention on Opus candidates but reduce it on Sonnet candidates, supporting separate Designer and Forecaster models with generator-dependent effects.

Table 5: **Selection across Designer and Forecaster models.** Top-1/Top-2 task-winner retention (%) at ten epochs, averaged equally over three tasks and six rounds per generator. Self denotes the generating model.

	Opus candidates		Sonnet candidates	
Forecaster	Top-1	Top-2	Top-1	Top-2
Best observed score	50.0	83.3	50.0	55.6
Self	50.0	77.8	50.0	72.2
DeepSeek V4.1 Flash	50.0	77.8	50.0	66.7
GPT-5.6 Sol	44.4	77.8	50.0	66.7
Gemini 3.8 Flash	44.4	83.3	55.6	61.1

4.4 Search cost and winner retention

To estimate screening savings, we retrospectively select up to $k \in \{1, \ldots, 6\}$ candidates per recorded round by combined forecast $\widehat{S}_a$ at epoch 10. Cost assumes that only selected candidates continue on all three tasks to epoch 40. Selecting two retains a full-budget winner in 83.3% of rounds and would save 54.9% of task-epochs; selecting three gives 91.7% retention and estimated savings of 44.9%, as shown in Table 6. Estimated GPU job-hours fall from 68.58 under full training to 31.86 and 38.85, respectively. The 36 forecasting calls use 1.167M input and 0.120M output tokens and are reused across retention budgets; accounting details appear in Supplementary A.2.

Table 6: **Winner retention and estimated screening cost.** Epoch-10 forecasts retrospectively select up to k candidates per round across 36 rounds. Completed 40-epoch runs provide ground truth; cost assumes only selected candidates continue. Retention and epoch savings are percentages.

Method	Winner retention	Full runs	Task-epochs	Epoch savings	GPU job-hours
Full training	100.0	807	32,280	0.0	68.58
PEEK ($k = 1$)	61.1	108	11,310	65.0	24.52
PEEK ($k = 2$)	83.3	216	14,550	54.9	31.86
PEEK ($k = 3$)	91.7	324	17,790	44.9	38.85
PEEK ($k = 4$)	94.4	429	20,940	35.1	45.53
PEEK ($k = 5$)	97.2	534	24,090	25.4	52.18
PEEK ($k = 6$)	100.0	633	27,060	16.2	58.27

5 Conclusion

AutoBCI combines PGAD for architecture discovery across motor imagery, emotion recognition, and sleep staging with PEEK for early performance forecasting. Across six LLM-guided searches, every search improves its best validation score. The strongest discovered architecture reaches 64.16% average test bAcc, 0.29 pp above REVE, which leads on wF1 and Cohen's Kappa. After ten epochs, PEEK reduces forecast MAE by 38.1%, although its top-two selection retains fewer combined-score winners than the best-observed-score baseline.

AI USE DISCLOSURE

Generative AI is part of the research method: the six LLMs in Table 2 propose and refine executable EEG architectures from task specifications and validation feedback, and predict full-budget validation performance from early training evidence. Generated architectures undergo the implementation checks described in Supplementary A.6; their reported decoding results are computed by training and evaluating the models on the stated data partitions. We also used AI assistants to do literature review, and text polishing. The authors take responsibility for the final manuscript, experimental claims, and accompanying artifacts.

ETHICS STATEMENT

This study uses publicly available EEG datasets and involves no participant recruitment or new human-data collection. All experiments consist of computational analyses of existing recordings. We identify no additional ethical concerns arising from these analyses.

REPRODUCIBILITY STATEMENT

The main paper and supplementary material describe the datasets, data splits, search procedure, training settings, and evaluation protocol. We provide the forecasting prompts and selected architecture definitions in the supplement. We will release the code, experiment configurations, and supporting materials needed to reproduce all results reported in this paper.

REFERENCES

Diego Alvarez-Estevez and Roselyne M Rijsman. Inter-database validation of a deep learning approach for automatic sleep scoring. *PloS one*, 16(8):e0256111, 2021.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

Angelica Chen, David Dohan, and David So. Evoprompting: Language models for code-level neural architecture search. *Advances in neural information processing systems*, 36:7787–7817, 2023.

Hohyun Cho, Minkyu Ahn, Sangtae Ahn, Moonyoung Kwon, and Sung Chan Jun. Eeg datasets for motor imagery brain–computer interface. *GigaScience*, 6(7):gix034, 2017.

Yanna Ding, Zijie Huang, Xiao Shou, Yihang Guo, Yizhou Sun, and Jianxi Gao. Architecture-aware learning curve extrapolation via graph ordinary differential equation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 39, pp. 16289–16297, 2025.

Yi Ding, Neethu Robinson, Su Zhang, Qiuhao Zeng, and Cuntai Guan. Tsception: Capturing temporal dynamics and spatial asymmetry from eeg for emotion recognition. *IEEE Transactions on affective computing*, 14(3):2238–2250, 2022.

Tobias Domhan, Jost Tobias Springenberg, Frank Hutter, et al. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, volume 15, pp. 3460–8, 2015.

Ruo-Nan Duan, Jia-Yi Zhu, and Bao-Liang Lu. Differential entropy feature for eeg-based emotion classification. In *2013 6th international IEEE/EMBS conference on neural engineering (NER)*, pp. 81–84. IEEE, 2013.

Yiqun Duan, Zhen Wang, Yi Li, Jianhang Tang, Yu-Kai Wang, and Chin-Teng Lin. Cross task neural architecture search for eeg signal recognition. *Neurocomputing*, 545:126260, 2023.

Yassine El Ouahidi, Jonathan Lys, Philipp Thölke, Nicolas Farrugia, Bastien Pasdeloup, Vincent Gripon, Karim Jerbi, and Giulia Lioi. Reve: A foundation model for eeg-adapting to any setup with large-scale pretraining on 25,000 subjects. *Advances in Neural Information Processing Systems*, 38:22541–22577, 2026.

Emadeldeen Eldele, Zhenghua Chen, Chengyu Liu, Min Wu, Chee-Keong Kwoh, Xiaoli Li, and Cuntai Guan. An attention-based deep learning approach for sleep stage classification with single-channel eeg. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 29:809–818, 2021.

Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International conference on machine learning*, pp. 1437–1446. PMLR, 2018.

Dengzhe Hou, Lingyu Jiang, Fangzhou Lin, and Kazunori D Yamada. Cogeegagent: Toward autonomous cognitive eeg analysis with grounded execution and selection-aware verification. *arXiv preprint arXiv:2607.25045*, 2026.

Ji-Hoon Jeong, Jeong-Hyun Cho, Young-Eun Lee, Seo-Hyun Lee, Gi-Hwan Shin, Young-Seok Kweon, José del R Millán, Klaus-Robert Müller, and Seong-Whan Lee. 2020 international brain–computer interface competition: A review. *Frontiers in human neuroscience*, 16:898300, 2022.

Muyun Jiang, Shuailei Zhang, Zhenjie Yang, Mengjun Wu, Weibang Jiang, Zhiwei Guo, Wei Zhang, Rui Liu, Shangen Zhang, Yong Li, et al. Leaf: Language-eeg aligned foundation model for brain-computer interfaces. *arXiv preprint arXiv:2509.24302*, 2025.

Muyun Jiang, Wei Zhang, Yi Ding, Kok Ann Colin Teo, LaiGuan Fong, Shuailei Zhang, Zhiwei Guo, Chenyu Liu, Raghavan Bhuvanakantham, Wei Khang Jeremy Sim, et al. Decoding covert speech from eeg by functional areas spatio-temporal transformer. *IEEE Journal of Biomedical and Health Informatics*, 2026.

Wei-Bang Jiang, Liming Zhao, and Bao-Liang Lu. Large brain model for learning generic representations with tremendous eeg data in bci. In *International Conference on Learning Representations*, volume 2024, pp. 16405–16426, 2024.

Bob Kemp, Aeilko H Zwinderman, Bert Tuk, Hilbert AC Kamphuisen, and Josefien JL Oberye. Analysis of a sleep-dependent neuronal feedback loop: the slow-wave microcontinuity of the eeg. *IEEE Transactions on Biomedical Engineering*, 47(9):1185–1194, 2000.

Sirvan Khalighi, Teresa Sousa, José Moutinho Santos, and Urbano Nunes. Isruc-sleep: A comprehensive public dataset for sleep researchers. *Computer methods and programs in biomedicine*, 124: 180–192, 2016.

Vernon J Lawhern, Amelia J Solon, Nicholas R Waytowich, Stephen M Gordon, Chou P Hung, and Brent J Lance. Eegnet: a compact convolutional neural network for eeg-based brain–computer interfaces. *Journal of neural engineering*, 15(5):056013, 2018.

Min-Ho Lee, O-Yeon Kwon, Yong-Jeong Kim, Hong-Kyung Kim, Young-Eun Lee, John Williamson, Siamac Fazli, and Seong-Whan Lee. Eeg dataset and openbmi toolbox for three bci paradigms: An investigation into bci illiteracy. *GigaScience*, 8(5):giz002, 2019.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of machine learning research*, 18(185):1–52, 2018.

Wei Liu, Jie-Lin Qiu, Wei-Long Zheng, and Bao-Liang Lu. Comparing recognition performance and robustness of multimodal deep learning models for multimodal emotion recognition. *IEEE Transactions on Cognitive and Developmental Systems*, (2):715–729, 2021.

Wuche Liu, Yiran Qiao, Linlin Hou, Rui Yang, Shusen Pu, Song Wang, and Jing Ma. Ns-copilot: An llm-driven agent system for autonomous neuroscience analysis. *arXiv preprint arXiv:2609.01971*, 2026.

Jun Ma, Banghua Yang, Wenzheng Qiu, Yunzhe Li, Shouwei Gao, and Xinxing Xia. A large eeg dataset for studying cross-session variability in motor imagery brain-computer interface. *Scientific Data*, 9(1):531, 2022.

Gerwin Schalk, Dennis J McFarland, Thilo Hinterberger, Niels Birbaumer, and Jonathan R Wolpaw. Bci2000: a general-purpose brain-computer interface (bci) system. *IEEE Transactions on biomedical engineering*, 51(6):1034–1043, 2004.

Robin Tibor Schirrmeister, Jost Tobias Springenberg, Lukas Dominique Josef Fiederer, Martin Glasstetter, Katharina Eggensperger, Michael Tangermann, Frank Hutter, Wolfram Burgard, and Tonio Ball. Deep learning with convolutional neural networks for eeg decoding and visualization. *Human brain mapping*, 38(11):5391–5420, 2017.

Jaeyoung Shin, Alexander von Lühmann, Benjamin Blankertz, Do-Won Kim, Jichai Jeong, Han-Jeong Hwang, and Klaus-Robert Müller. Open access dataset for eeg+ nirs single-trial classification. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 25(10):1735–1745, 2016.

Yonghao Song, Qingqing Zheng, Bingchuan Liu, and Xiaorong Gao. Eeg conformer: Convolutional transformer for eeg decoding and visualization. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 31:710–719, 2022.

Akara Supratak, Hao Dong, Chao Wu, and Yike Guo. Deepsleepnet: A model for automatic sleep stage scoring based on raw single-channel eeg. *IEEE transactions on neural systems and rehabilitation engineering*, 25(11):1998–2008, 2017.

Michael Tangermann, Klaus-Robert Müller, Ad Aertsen, Niels Birbaumer, Christoph Braun, Clemens Brunner, Robert Leeb, Carsten Mehring, Kai J Miller, Gernot R Müller-Putz, et al. Review of the bci competition iv. *Frontiers in neuroscience*, 6:55, 2012.

Guoan Wang, Shihao Yang, and Feng Liu. Neuroweaver: An autonomous evolutionary agent for exploring the programmatic space of eeg analysis pipelines. *arXiv preprint arXiv:2602.13473*, 2026.

Jiquan Wang, Sha Zhao, Zhiling Luo, Yangxuan Zhou, Haiteng Jiang, Shijian Li, Tao Li, and Gang Pan. Cbramod: A criss-cross brain foundation model for eeg decoding. In *International conference on learning representations*, volume 2025, pp. 75310–75346, 2025.

Fang Weilin, Yu Xue, Yuan Lilian, Mohammad Kamrul Hasan, and Khursheed Aurangzeb. Large language model assisted evolutionary neural architecture search with population knowledge base enhancement. *Information Sciences*, pp. 123110, 2026.

Zekang Yang, Wang Zeng, Sheng Jin, Chen Qian, Ping Luo, and Wentao Liu. Nader: Neural architecture design via multi-agent collaboration. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4452–4461. IEEE, 2025.

Sha Zhao, Mingyi Peng, Haiteng Jiang, Tao Li, and Shijian Li. Eeg agent: A unified framework for automated eeg analysis using large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 18063–18071, 2026.

Mingkai Zheng, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. Can gpt-4 perform neural architecture search? *arXiv preprint arXiv:2304.10970*, 2023.

Wei-Long Zheng, Wei Liu, Yifei Lu, Bao-Liang Lu, and Andrzej Cichocki. Emotionmeter: A multimodal framework for recognizing human emotions. *IEEE transactions on cybernetics*, 49(3): 1110–1122, 2018.

A SUPPLEMENTARY MATERIAL

CONTENTS

A.1 Related Work 13
A.1.1 EEG Representation Learning and Architecture Design 13
A.1.2 LLM-Guided Architecture Search and Early Performance Prediction 14
A.2 Evaluation details and supplementary comparisons 14
A.3 Round-by-round architecture details 15
A.4 Baseline training and evaluation 16
A.5 Data preparation and split construction 16
A.5.1 Dataset descriptions 16
A.5.2 Task-specific split assignments 18
A.5.3 Signal representation and label handling 18
A.5.4 Dataset split rules and sizes 19
A.6 Search implementation and execution details 19
A.7 Candidate-training settings 20
A.8 Training precision: computational cost and predictive performance 21
A.9 Winner Architecture 22
A.9.1 Claude Sonnet 5 22
A.9.2 DeepSeek V4.1 Flash 23
A.9.3 Gemini 3.8 Flash 24
A.9.4 Qwen3.8 Flash 25
A.9.5 GPT-5.6 Sol 26
A.9.6 Claude Opus 5.5 26
A.10 Complete prompts and structured interfaces 27
A.10.1 PGAD: complete first-round prompt 27
A.10.2 PGAD: subsequent-round prompt 29
A.10.3 PEEK: complete forecasting prompt template 30

A.1 RELATED WORK

A.1.1 EEG REPRESENTATION LEARNING AND ARCHITECTURE DESIGN

EEG architectures encode assumptions about temporal structure and spatial organization. EEGNet uses depthwise and separable convolutions to construct a compact architecture evaluated across several BCI paradigms (Lawhern et al., 2018). Task-specific designs introduce further structure: TSception combines temporal filters at multiple scales with spatial filters that capture hemispheric asymmetry for emotion recognition (Ding et al., 2022), while DeepSleepNet combines convolutional feature extraction with recurrent modeling of sleep-stage dependencies (Supratak et al., 2017). Foundation models study transferable EEG representations through pretraining. LaBraM learns discrete neural tokens through neural-spectrum prediction and pretrains a Transformer using masked token prediction (Jiang et al., 2024). CBraMod uses criss-cross attention to model spatial and temporal dependencies in heterogeneous EEG recordings (Wang et al., 2025). LEAF incorporates task instructions and aligns EEG representations with language semantics across tasks and label spaces (Jiang et al., 2025).

AutoBCI selects architecture code using combined validation scores from emotion recognition, motor imagery, and sleep staging, with independent training and separate parameters for each task family.

Architecture search also has direct precedents in EEG decoding. CTNAS-EEG introduces a search space compatible with multiple EEG tasks and a constrained search procedure, and examines architectural variation across tasks and subjects (Duan et al., 2023). PGAD uses executable model proposals and an accumulated pool ranked by an equal-weight aggregate of task scores. Task-specific learning curves and validation results accompany each selected parent, providing feedback for subsequent code-level refinements.

A.1.2 LLM-Guided Architecture Search and Early Performance Prediction

Language models can serve as proposal operators in architecture search. EvoPrompting uses code-generating language models for evolutionary mutation and crossover, together with evolutionary prompt construction and soft prompt tuning (Chen et al., 2023). GENIUS treats GPT-4 as a black-box optimizer that proposes and iteratively refines architectures (Zheng et al., 2023). NADER coordinates specialized agents around a graph representation of architectures and uses reflection on feedback and prior experience to guide modifications (Yang et al., 2025). PGAD combines parent refinements and fresh proposals with a shared selection objective across independently trained EEG task models.

Reducing the cost of candidate evaluation is a related problem. Domhan et al. (2015) extrapolate partial learning curves to terminate unpromising runs. Baker et al. (2017) combine architecture features, hyperparameters, and partial validation trajectories to predict performance during architecture search. Architecture-aware learning-curve extrapolation also models network structure through a graph ordinary differential equation (Ding et al., 2025). Resource-allocation methods such as Hyperband use successive halving to distribute training budgets (Li et al., 2018); BOHB combines this allocation strategy with model-based configuration selection (Falkner et al., 2018).

A.2 Evaluation details and supplementary comparisons

Forecasting inputs and aggregation. In the primary analysis, each generating LLM forecasts its own candidates in isolated sessions with tools disabled. Inputs contain the training protocol, parameter counts, training losses, and validation prefixes for all three tasks. Both the code-and-curve and curve-only conditions use the same anonymous candidate order and early evidence; only the former includes architecture code, with comments and docstrings removed. Candidate names, dataset names, and parent identities are omitted. The target for each task is the maximum, within 40 epochs, of its equal-dataset mean validation bAcc. Combined scores average these task maxima equally, allowing different checkpoint epochs across tasks.

The cross-model comparison in Table 5 reuses the frozen ten-epoch code-and-curve prompts, anonymous ordering, and self-forecasts from the Opus and Sonnet searches. All 36 additional calls succeed (three forecasters across 12 rounds), with one forecast per forecaster and round. Generator identities are omitted from the prompts, and later validation epochs and test results remain unavailable to the forecasters.

Nominal screening budget. Observing all eight candidates for ten epochs and continuing two to epoch 40 uses $8 \times 10 + 2 \times (40 - 10) = 140$ candidate-epochs per task, compared with $8 \times 40 = 320$ under full training. The resulting 56.25% reduction excludes forecasting overhead and failed training attempts.

Retention–cost accounting. The fixed cohort in Table 6 uses all 36 successfully paired rounds at ten epochs, with 269 candidates and three independently trained tasks per candidate. For a round with n_r candidates, keeping $k_r = \min(k, n_r)$ gives $3[10n_r + 30k_r]$ task-epochs. Full training uses 32,280 task-epochs. The actual cohort has fewer than eight completed candidates in some rounds, giving 54.9% savings at $k = 2$, compared with 56.25% for an idealized eight-candidate round. Predicted ties retain the original anonymous candidate order; any tied true maximum counts as a winner. These estimates evaluate screening on the recorded candidate batches. How screening changes parent selection, subsequent proposals, and final search performance remains to be evaluated.

Recorded 40-epoch task durations include validation. Estimated GPU job-hours sum these durations for continued candidates and one quarter of each duration for candidates stopped at ten epochs, assuming constant average epoch time. These are accumulated job durations, not physical GPU occupancy or elapsed search time; concurrent jobs can share a GPU. Token counts sum saved generation and accepted forecasting calls, including input cache reads and writes once. The common generation budget is 0.281M input and 0.501M output tokens; PEEK adds 1.167M input and 0.120M output tokens.

A.3 Round-by-round architecture details

Supplementary Figure 6 identifies the incumbent architecture after each round of the six searches shown in Figure 3.

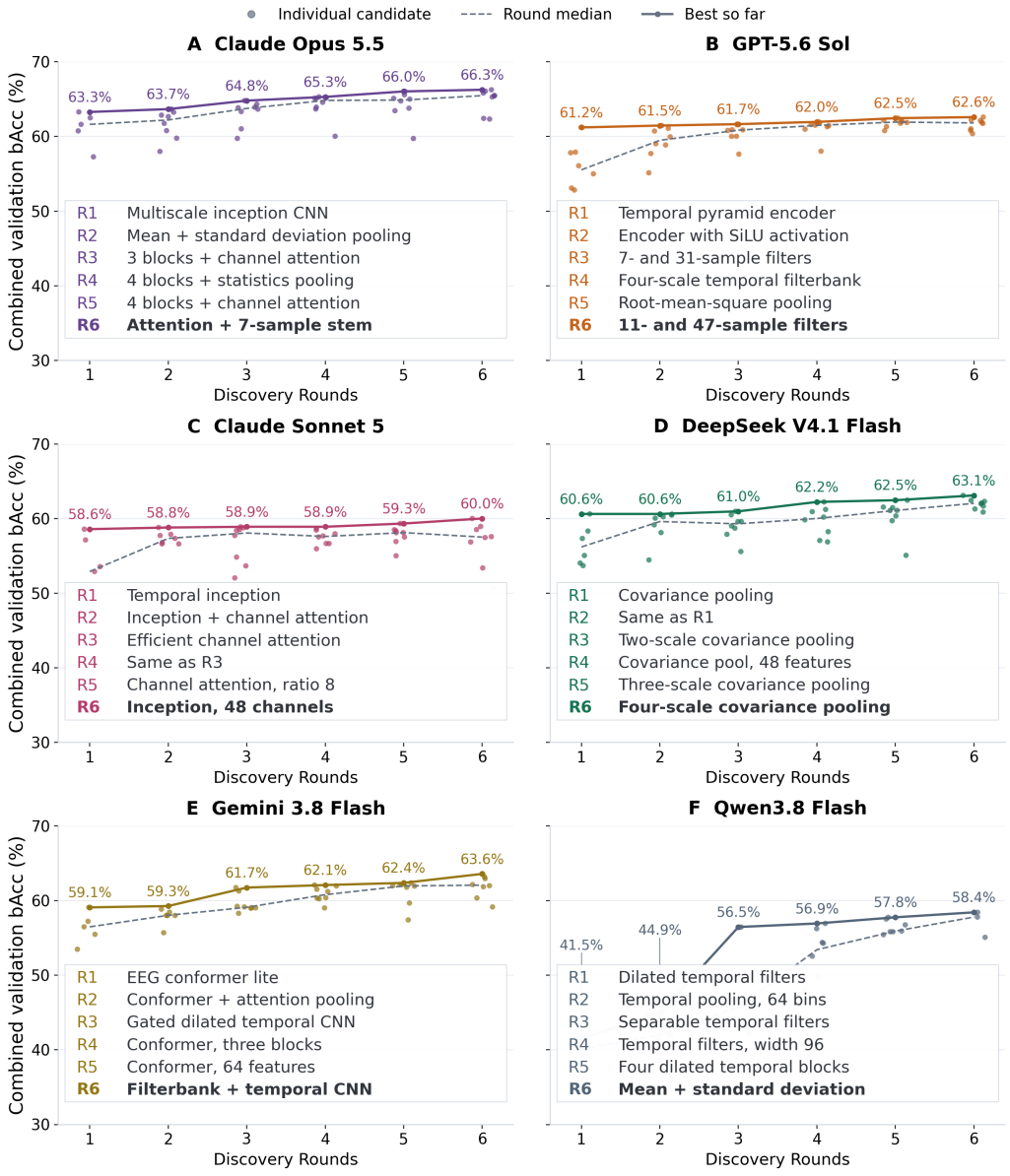


Figure 6: **Architecture discovery with round incumbents.** Dots show individual evaluated candidates; dashed and solid lines show round medians and cumulative best bAcc. Each inset names the incumbent after rounds R1–R6; bold text marks the final architecture.

A.4 BASELINE TRAINING AND EVALUATION

To interpret the held-out comparison, we describe how each baseline is initialized, adapted to the task inputs, and optimized. Each model is trained independently on the pooled datasets of each task family, using the same harmonized labels, normalization, fixed partitions, and sleep subsets as AutoBCI. Each epoch visits all training examples in shuffled order, including the final incomplete batch. Validation bAcc is averaged equally across datasets after every epoch; a checkpoint is replaced only when this score increases. The selected checkpoint is evaluated on the held-out test partitions, and the three task scores receive equal weight in the combined results.

Optimization and training budgets. EEGNet, DeepConvNet, TSception, EEG-Conformer, AttnSleep, and FAST are initialized from scratch and trained for 100 epochs with batch size 512 and initial learning rate 10^{-3}. LaBraM, CBraMod, LEAF, and REVE use a configured budget of 40 epochs, batch size 256 without gradient accumulation, and initial learning rate 10^{-4}. All main-comparison runs use three random seed averaged, fused AdamW with weight decay 10^{-4}, coefficients $(0.9, 0.999)$, numerical epsilon 10^{-8}, and global gradient-norm clipping at 5.0. Cosine annealing decreases the learning rate toward zero over the configured epoch budget. Training uses unweighted cross-entropy computed from FP32 logits, with model parameters and AdamW moments in BF16.

Pretrained initialization and task heads. LaBraM-base loads the student encoder from `labram-base.pth`, with a newly initialized mean-pooling normalization and classification head; pretraining-only prediction components are omitted. CBraMod loads `pretrained_weights.pth`, replaces its reconstruction projection with an identity mapping, and adds a linear classifier to the mean-pooled representations. REVE-base loads the official `brain-bzh/reve-base` checkpoint and adds a linear classifier after averaging channel and time representations; electrode coordinates come from its official position bank. LEAF uses `leaf-v1.0-instruct-mpnet-base.ckpt` for MI and emotion, with its EEG tower and a new flattened tower-token classifier trained jointly. The QFormer is unused by this classifier. For these pretrained conditions, the retained encoder and task head are both fine-tuned. LEAF on sleep is initialized from scratch, with its channel count and temporal capacity adapted to the native sleep input.

Input adaptations. Conventional baselines receive the native task tensors: 65×800 for MI and emotion and 8×3000 for sleep. The foundation baselines also retain the 65×800 MI and emotion inputs at 200 Hz. For sleep, LaBraM, CBraMod, and REVE linearly resample each complete 30-second example from 100 to 200 Hz and divide it into three non-overlapping ten-second windows. Window logits are averaged before computing the example-level loss and metrics, preserving the original example and split identities. LaBraM uses named electrode indices and 200-sample patches. LEAF sleep instead consumes the complete native 8×3000 tensor directly. CBraMod computes its FFT in FP32 before casting spectral features back to BF16; REVE similarly retains FP32 Fourier-coordinate calculations and normalization reductions while adapting linear-layer inputs to the model dtype.

A.5 DATA PREPARATION AND SPLIT CONSTRUCTION

Task-wise training and evaluation. For each candidate architecture, we pool only the training splits within each task family and train one model from scratch for 40 epochs. The three task models share architecture code but learn separate weights and output heads. The class counts in Table 1 refer to the union of semantic labels within each family: seven for emotion, seven for MI, and five for sleep. Validation and test splits remain separate from training, and each dataset is evaluated individually. Architecture selection uses validation scores averaged equally across datasets within each task, then equally across the three tasks. Test splits are used only for final evaluation.

A.5.1 DATASET DESCRIPTIONS

The pooled task families combine datasets with different acquisition protocols and participant populations. The following list describes each source and the version used in our experiments;

emotion and MI follow the LEAF dataset preparation documentation (Jiang et al., 2025). Split assignments and retained example counts are given below in Tables 7 and 8.

Emotion recognition. The three SEED datasets contain EEG recorded during emotion-eliciting video trials, with different emotion categories and session structures.

- **SEED** (Duan et al., 2013) contains 62-channel recordings from 15 participants across three sessions, with 15 trials per session and three labels: Negative, Neutral, and Positive. The LEAF version concatenates corresponding trials across sessions before extracting four-second windows.
- **SEED-IV** (Zheng et al., 2018) contains 62-channel recordings from 15 participants across three sessions, each with 24 video trials. Four-second windows retain four emotion labels: Neutral, Sad, Fear, and Happy.
- **SEED-V** (Liu et al., 2021) contains recordings from 16 participants across three sessions, each with 15 video trials. After removing ocular and mastoid channels, 62 EEG channels are segmented into four-second windows with Disgust, Fear, Sad, Neutral, and Happy labels.

Motor imagery. The motor-decoding collection covers hand, foot, tongue, and grasp-related tasks. The descriptions below distinguish the source recordings from the classes retained by the LEAF preparation.

- **BCI Competition IV-2a** (Tangermann et al., 2012) provides 22-channel EEG from nine participants performing left-hand, right-hand, foot, and tongue imagery. Both acquisition-session files are included, and the preparation retains a four-second interval from each trial.
- **BCI Upper Limb** (Jeong et al., 2022) contains recordings from 15 participants imagining three grasp types: cylindrical, spherical, and lumbrical. The published training and validation recordings are combined before applying the subject partitions used here; each example covers the final four-second imagery stage.
- **Cho2017** (Cho et al., 2017) is a left- versus right-hand imagery dataset. The LEAF export contains 49 participants after excluding subjects 32, 46, and 49; its preprocessing appends 200 samples by edge padding before constructing the common input representation.
- **HighGamma** (Schirrmeister et al., 2017) contributes recordings from 14 participants. The LEAF preparation initially retains left-hand, right-hand, and feet classes, and the experimental loader further selects the two hand classes for binary decoding.
- **OpenBMI** (Lee et al., 2019) contributes left- and right-hand imagery recordings from 54 participants across two acquisition sessions. The LEAF version combines the training and test recordings from both sessions for each participant before applying the experiment's subject partitions.
- **PhysioNet** (Schalk et al., 2004) contains 64-channel recordings from 109 participants performing real and imagined movements. The LEAF preparation processes runs R03–R14 and retains the two non-rest event codes; their source meanings depend on the run (left/right fist or both fists/both feet), while the stored task catalog names the two classes Left and Right.
- **ShanghaiU** (Ma et al., 2022) contains left- and right-hand imagery recordings from 25 participants over five sessions. All five sessions are combined per participant, and the two reference channels A1 and A2 are removed before mapping to the common montage.
- **Shin2017A** (Shin et al., 2016) contributes the EEG component of an EEG–NIRS motor-imagery dataset with left- and right-hand labels. The LEAF version contains 28 available participants (subject S05 is unavailable), uses three imagery blocks per participant, and removes the ocular channels before extracting four-second epochs.

Sleep staging. The sleep datasets provide overnight polysomnography with expert stage annotations. We use their EEG signals to classify 30-second epochs as Wake, N1, N2, N3, or REM.

- **Sleep-EDF** (Kemp et al., 2000) uses the expanded database's Sleep Cassette cohort: 153 recordings from 78 participants in a study of age-related sleep changes. The original EEG derivations are Fpz–Cz and Pz–Oz at 100 Hz, with Rechtschaffen–Kales stage annotations. Our preparation merges stages 3 and 4 into N3 and retains up to 30 minutes of wakefulness before and after sleep.[1]
- **HMC** (Alvarez-Estevez & Rijsman, 2021) contains 151 overnight recordings from patients referred for sleep evaluation at Haaglanden Medisch Centrum in the Netherlands. Four EEG derivations (F4/M1, C4/M1, O2/M1, and C3/M2) were acquired at 256 Hz, and sleep technicians scored the recordings using AASM guidelines. Our preparation uses the original EDF release and resamples EEG to 100 Hz.[2]
- **ISRUC** (Khalighi et al., 2016) uses subgroup I, comprising one overnight recording per subject from 100 adults with sleep disorders. The dataset provides annotations from two experts; our preparation uses scorer 1, excludes subject 55 because of conflicting annotations, and retains 99 subjects. The available EEG derivations are mapped to the shared eight-channel sleep representation.[3]

A.5.2 Task-specific split assignments

Emotion recognition. We use the four-second HDF5 versions of SEED, SEED-IV, and SEED-V, preserving their predefined training, validation, and test arrays. The preparation scripts assign stimulus trials within subjects to these splits: SEED uses trials 1–9, 10–12, and 13–15; SEED-IV uses trials 1–16, 17–20, and 21–24 in each session; SEED-V uses trials 1–5, 6–10, and 11–15 in each session. AutoBCI reads the stored assignments without drawing a new validation subset.

Motor imagery. The MI files store examples by subject. In the HDF5 group iteration order, the first 7, 11, 40, 10, 42, 80, 20, and 22 subjects form the development partitions of IV-2a, Upper Limb, Cho2017, HighGamma, OpenBMI, PhysioNet, ShanghaiU, and Shin2017A, respectively. The remaining 2, 4, 9, 4, 12, 29, 5, and 6 subjects form their test partitions. Within each development partition, examples are concatenated in HDF5 subject order after label filtering. Validation takes zero-based example indices $0, 5, 10, \ldots$ (approximately 20%), and training takes the remaining examples. Training and validation therefore share subjects, while test subjects are disjoint from both. HighGamma retains only local labels 0 and 1, corresponding to the catalog's left- and right-hand classes. The PhysioNet-MI recordings were collected with the BCI2000 system (Schalk et al., 2004).

Sleep staging. Sleep-EDF, HMC, and ISRUC use five classes: Wake, N1, N2, N3, and REM. We preserve their stored subject-disjoint split manifests, with 49/13/16 subjects for Sleep-EDF, 96/24/31 for HMC, and 63/16/20 for ISRUC in training/validation/test. Sleep-EDF uses the cassette cohort; ISRUC uses scorer 1 and excludes subject 55. To define the search workload, the stored files retain 25%, 40%, and 50% of the examples in Sleep-EDF, HMC, and ISRUC, respectively. Subsampling is uniform without replacement within each split, and retains the floor of the original split size multiplied by its retention fraction. The retained subsets are fixed throughout architecture search and final evaluation.

A.5.3 Signal representation and label handling

Emotion and MI examples use a common 65-channel representation with 800 samples at 200 Hz. The LEAF preparation scripts map source electrodes to the `LEAF-ch65` template and apply percentile clipping and robust scaling before export. Emotion preparation uses a 0.1–70 Hz bandpass and a 50 Hz notch filter. MI preparation uses a 0.3–40 Hz bandpass, except Upper Limb, which uses 0.1–70 Hz. The search consumes the exported arrays; signal filtering and electrode mapping are fixed before candidate training.

Sleep examples preserve complete 30-second epochs at 100 Hz. The ordered channel slots are Fpz, F3, F4, C3, C4, Pz, O1, and O2. The stored preprocessing uses a 0.5–40 Hz bandpass, rejects epochs with excessive absolute amplitude above $2{,}000\,\mu$V before filtering, and applies per-recording clipping

[1] `https://physionet.org/content/sleep-edfx/1.0.0/`
[2] `https://physionet.org/content/hmc-sleep-staging/1.1/`
[3] `https://sleeptight.isr.uc.pt/`

at the 0.1th and 99.9th percentiles followed by median/IQR scaling. Missing channel slots are interpolated using an MNE spherical-spline mapping, retaining the original references of measured derivations.

For every task, the training loader additionally standardizes each example and channel along time:

$$\widetilde{x}_{c,t} = \frac{x_{c,t} - \mu_c}{\max(\sigma_c, 10^{-6})}, \tag{6}$$

where μ_c and σ_c are the temporal mean and population standard deviation of that channel in the example. Normalized inputs are cached in CPU memory as BF16 tensors. This normalization uses statistics from the example itself. The configured files already match their task's temporal input length, so the loader passes all 800 or 3,000 samples to the model.

Each task uses one output head spanning the union of its catalog labels. Labels with identical names within a task share an output class. This produces seven emotion outputs (Positive, Neutral, Negative, Sad, Fear, Happy, and Disgust), seven MI outputs (Left, Right, Foot, Tongue, Cylin, Sphe, and Lumbrical), and five sleep outputs. During validation and testing, logits for classes absent from the example's source dataset are set to negative infinity before prediction. Dataset identity is used for this evaluation mask and metric aggregation; the model receives only the EEG tensor.

A.5.4 Dataset split rules and sizes

The split unit determines which examples and subjects are held out during evaluation. Emotion uses predefined stimulus-trial assignments within subjects, MI holds out test subjects and divides development examples into training and validation, and sleep separates subjects across all three partitions. The dataset-specific assignments are summarized in Table 7; the resulting example counts after label filtering and sleep subsampling appear in Table 8.

Table 7: **Dataset split construction.** Emotion trial numbers are one-based and apply within each subject and session; subjects occur in all three splits. MI subject ranges are zero-based positions in HDF5 group iteration order, not original subject IDs. MI training and validation use approximately 80% and 20% of examples from the same development subjects, respectively; test subjects are held out. Sleep entries give numbers of mutually disjoint subjects.

Task	Dataset	Train	Validation	Test
Emotion	SEED	Trials 1–9	Trials 10–12	Trials 13–15
	SEED-IV	Trials 1–16	Trials 17–20	Trials 21–24
	SEED-V	Trials 1–5	Trials 6–10	Trials 11–15
MI	BCI Competition IV-2a	Subjects 0–6	Subjects 0–6	Subjects 7–8
	BCI Upper Limb	Subjects 0–10	Subjects 0–10	Subjects 11–14
	Cho2017	Subjects 0–39	Subjects 0–39	Subjects 40–48
	HighGamma	Subjects 0–9	Subjects 0–9	Subjects 10–13
	OpenBMI	Subjects 0–41	Subjects 0–41	Subjects 42–53
	PhysioNet	Subjects 0–79	Subjects 0–79	Subjects 80–108
	ShanghaiU	Subjects 0–19	Subjects 0–19	Subjects 20–24
	Shin2017A	Subjects 0–21	Subjects 0–21	Subjects 22–27
Sleep	Sleep-EDF	49 subjects	13 subjects	16 subjects
	HMC	96 subjects	24 subjects	31 subjects
	ISRUC	63 subjects	16 subjects	20 subjects

A.6 Search implementation and execution details

Candidates use PyTorch and expose `build_model(n_channels, n_samples, n_classes)`. The search prompt permits PyTorch and Python's `math` module, excludes recurrent layers and custom temporal recurrence, and requires all trainable parameters to be created before the forward pass. Models are trained from scratch with the common training procedure below.

Complete prompt templates and structured interfaces appear in Supplementary A.10. Each generation call produces the entire batch of eight proposals as a structured JSON response. The harness runs in a

Table 8: **Dataset composition and split sizes.** Counts refer to EEG examples, including 30-second epochs for sleep. Classes are the retained local classes in each dataset. Emotion and MI inputs have shape 65×800; sleep inputs have shape 8×3000.

Task	Dataset	Classes	Train	Validation	Test
Emotion	SEED (3 classes)	3	22,545	7,905	7,620
	SEED-IV	4	26,025	5,340	6,210
	SEED-V	5	8,512	10,672	9,984
MI	BCI Competition IV-2a	4	3,148	788	1,152
	BCI Upper Limb	3	2,640	660	1,200
	Cho2017	2	6,464	1,616	1,800
	HighGamma	2	3,761	941	2,040
	OpenBMI	2	13,440	3,360	4,800
	PhysioNet	2	11,447	2,862	5,262
	ShanghaiU	2	7,712	1,929	2,347
	Shin2017A	2	1,056	264	360
Sleep	Sleep-EDF	5	30,062	9,237	9,567
	HMC	5	34,764	8,688	11,422
	ISRUC	5	28,181	7,271	8,914

temporary working directory with tool use disabled. The supplied prompt contains the architectural constraints, training settings, parameter budget, and class categories. Later rounds also receive the two parents' source code, task-level validation curves, best epochs, parameter counts, and per-dataset validation scores at the selected epochs. Per-dataset results use anonymous identifiers in the prompt. Generation calls receive the measured parent evidence explicitly rather than retaining a conversational session across rounds.

The Claude Code and Gemini CLI adapters configure a maximum output of 65,536 tokens. The Codex adapter enforces the response schema through its structured output interface and leaves the output-token limit to the harness. Temperature and sampling seed are not explicitly set by the adapters. Recorded generation timeouts are 1,200 seconds in round 1 for all six searches. Rounds 2–6 use 1,200 seconds for DeepSeek and 600 seconds for the other five models. Each call records the requested model, provider, harness command, duration, response, and available model-identity evidence.

Before training, static checks validate the model interface and allowed imports. Numerical checks verify output shape, finite logits, consistent repeated evaluation, and fixed parameter registration for batch sizes one and two. Two synthetic optimization steps then check execution at the configured training batch size and precision. Actual training constructs a fresh model after these checks. Candidate implementation failures are recorded with their task and failure stage, and the generated source is preserved. Only candidates that complete all three tasks enter parent and champion selection.

A.7 Candidate-training settings

The shared optimization settings in Table 9 specify how candidates are trained under the protocol in Section 3.3.

An epoch visits every training example in the task's pooled datasets once, in a new shuffled order, including the final incomplete batch. Sampling is uniform over examples, so a dataset's contribution to the training loss scales with its number of examples. Validation is performed after each epoch and weights datasets equally according to Equation (1). The loss is unweighted cross-entropy; the training loop applies neither class reweighting nor data augmentation.

The seed initializes Python, NumPy, PyTorch, CUDA, and a separate generator for training-example shuffling. The best checkpoint is updated only when validation bAcc strictly increases, retaining the earliest epoch in a tie. The configured early observation budget for PEEK is ten epochs.

Training uses PyTorch `2.10.0+cu128` and NVIDIA RTX PRO 6000 Blackwell GPUs. Each task runs as a separate process on one GPU, and the coordinator schedules independent jobs according to GPU availability. Full BF16 training casts model parameters before optimizer construction, so AdamW moment tensors also use BF16. CUDA settings enable cuDNN benchmarking with

a benchmark limit of ten, TF32 for cuDNN, `medium` FP32 matrix-multiplication precision, and BF16 reduced precision reductions. Deterministic cuDNN execution is disabled. The recorded seed therefore specifies stochastic initialization and sampling without implying bitwise identical GPU execution. A controlled comparison of FP32, BF16 mixed, and full BF16 training appears in Supplementary A.8.

Table 9: **Common candidate-training settings.** The same settings apply to emotion, MI, and sleep in all six completed searches.

Setting	Value
Full training budget	40 epochs per task
Batch size	512
Optimizer	Fused AdamW
Initial learning rate	10^{-3}
Weight decay	10^{-4}
Adam coefficients (β_1, β_2)	$(0.9, 0.999)$
Numerical epsilon ϵ	10^{-8}
Learning-rate schedule	Cosine annealing; $T_{\max} = 40$, minimum 0
Gradient clipping	Global norm 5.0
Parameters, gradients, optimizer moments	BF16
Cross-entropy computation	FP32 logits and loss reduction
CPU threads per training process	2
Data cache	BF16 in CPU RAM
Preload batch size	512
Data-loader worker processes	0

A.8 Training precision: computational cost and predictive performance

We compare FP32, BF16 mixed precision, and full BF16 on BCI Competition IV-2a using the repository implementations of EEGNet (Lawhern et al., 2018), DeepConvNet (Schirrmeister et al., 2017), and the fixed Opus-discovered architecture. All models are trained from scratch on this single dataset. The existing split assigns 3,148 training and 788 validation examples to subjects A01–A07, and reserves 1,152 test examples from A08–A09. Inputs retain 65 channels and 800 time samples. For each architecture, the three modes share a common FP32 initialization before precision-specific casting, and the same minibatch order for seeds 0, 1, and 2. We train for 40 epochs with AdamW, learning rate 10^{-3}, weight decay 10^{-4}, cosine decay, and gradient clipping at norm 5. The batch size is 128. The earliest checkpoint with maximal validation bAcc is evaluated on the test set after all training runs finish.

FP32 uses FP32 parameters, gradients, and optimizer moments with TF32 disabled. BF16 mixed uses BF16 autocast while retaining FP32 parameters, gradients, and optimizer moments. Full BF16 casts model state and inputs to BF16 and stores gradients and AdamW moments in BF16, without FP32 master weights. All modes compute cross-entropy in FP32; explicit FP32 reductions in the generated architecture are preserved. Inputs are normalized and cached in FP32 for every condition, isolating training precision from cache quantization. Unlike the main search, this experiment uses batch size 128 and disables TF32 throughout.

Runs execute sequentially in fresh processes on one NVIDIA RTX PRO 6000 Blackwell GPU with PyTorch 2.10.0+cu128. Precision order rotates across seeds. Warmup exercises training and validation batch shapes, after which weights, optimizer state, and random-number generators are reset. CUDA-synchronized training time includes minibatch transfers and optimizer updates, excluding validation, preprocessing, warmup, profiling, and checkpoint writes. Memory is peak PyTorch allocation during training and validation. FLOPs count convolution and matrix-multiplication operations in forward and backward passes, with one multiply-add counted as two; normalization, elementwise operations, loss, and optimizer updates are excluded. These counts are unchanged by precision, whereas latency and memory depend on the numerical representation and kernel implementation.

Table 10: **Training precision on BCI Competition IV-2a.** Means and standard deviations over three seeds. GFLOPs are counted per training example (forward and backward); time is per training epoch; memory is mean peak allocation. Test bAcc uses validation-selected checkpoints.

Model	Precision	GFLOPs	Time (s)	MiB	Test bAcc (%)
EEGNet	FP32	0.126	0.650 ± 0.001	644	55.47 ± 0.69
	BF16 mixed	0.126	0.819 ± 0.009	371	54.48 ± 0.90
	BF16 full	0.126	0.810 ± 0.006	345	53.79 ± 1.11
DeepConvNet	FP32	0.328	0.380 ± 0.008	1336	58.91 ± 3.82
	BF16 mixed	0.328	0.427 ± 0.001	1099	57.96 ± 3.04
	BF16 full	0.328	0.423 ± 0.001	1083	55.90 ± 2.91
AutoBCI (Opus)	FP32	0.285	0.300 ± 0.001	351	55.99 ± 0.98
	BF16 mixed	0.285	0.336 ± 0.003	222	55.56 ± 0.83
	BF16 full	0.285	0.304 ± 0.003	208	56.80 ± 2.46

Lower precision consistently reduces memory, while its effects on predictive performance depend on the architecture, as shown in Table 10. Full BF16 reduces peak allocated memory relative to FP32 by 46.4% for EEGNet, 19.0% for DeepConvNet, and 40.8% for the Opus architecture. Mixed precision also reduces memory, but neither BF16 mode accelerates training in this setting: FP32 has the lowest mean epoch time for all three models, with full BF16 close to FP32 for Opus. Lower numerical precision therefore does not imply lower latency for these models at the fixed batch size.

Relative to FP32, full BF16 changes test bAcc by -1.68, -3.01, and $+0.81$ pp for EEGNet, DeepConvNet, and Opus, respectively. Mixed precision produces smaller absolute changes of -0.98, -0.95, and -0.43 pp. These three-seed results support reporting training precision as part of the experimental protocol: its memory advantage is consistent here, while its effect on predictive performance depends on the architecture. The comparison concerns precision within each model on one fixed dataset split.

A.9 Winner Architecture

To document the architectures behind the test results, we provide the model definitions of the six validation-selected champions from Table 3. Each listing contains the saved architecture code, including its auxiliary blocks and forward computation. The same code is instantiated independently for motor imagery, emotion recognition, and sleep staging, with task-specific input channels, class counts, and learned weights. The interface accepts inputs shaped $B \times C \times T$ and returns class logits.

A.9.1 Claude Sonnet 5

The Sonnet champion projects the input channels into 48 features and applies two multiscale convolution blocks with kernel sizes 3, 7, 15, and 31. Squeeze-and-excitation gates recalibrate the features before global average pooling and linear classification.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F


class InceptionBlock(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
        b = out_ch // 4
        rem = out_ch - 3 * b
        self.b1 = nn.Conv1d(in_ch, b, kernel_size=3, padding=1)
        self.b2 = nn.Conv1d(in_ch, b, kernel_size=7, padding=3)
        self.b3 = nn.Conv1d(in_ch, b, kernel_size=15, padding=7)
        self.b4 = nn.Conv1d(in_ch, rem, kernel_size=31, padding=15)
        self.bn = nn.BatchNorm1d(out_ch)

    def forward(self, x):
        y = torch.cat([self.b1(x), self.b2(x), self.b3(x), self.b4(x)], dim=1)
        return F.elu(self.bn(y))


```

```python
class SEBlock1d(nn.Module):
    def __init__(self, channels, reduction=8):
        super().__init__()
        red = max(channels // reduction, 4)
        self.fc1 = nn.Linear(channels, red)
        self.fc2 = nn.Linear(red, channels)

    def forward(self, x):
        s = x.mean(dim=-1)
        s = F.relu(self.fc1(s))
        s = torch.sigmoid(self.fc2(s))
        return x * s.unsqueeze(-1)


class InceptionTemporalSE(nn.Module):
    def __init__(self, n_channels, n_classes, hidden=48):
        super().__init__()
        self.spatial = nn.Conv1d(n_channels, hidden, kernel_size=1)
        self.block1 = InceptionBlock(hidden, hidden)
        self.se1 = SEBlock1d(hidden, reduction=8)
        self.pool1 = nn.MaxPool1d(2)
        self.block2 = InceptionBlock(hidden, hidden * 2)
        self.se2 = SEBlock1d(hidden * 2, reduction=8)
        self.pool2 = nn.AdaptiveAvgPool1d(1)
        self.classifier = nn.Linear(hidden * 2, n_classes)

    def forward(self, x):
        x = self.spatial(x)
        x = self.block1(x)
        x = self.se1(x)
        x = self.pool1(x)
        x = self.block2(x)
        x = self.se2(x)
        x = self.pool2(x).squeeze(-1)
        return self.classifier(x)


def build_model(n_channels: int, n_samples: int, n_classes: int):
    return InceptionTemporalSE(n_channels, n_classes)
```

A.9.2 DEEPSEEK V4.1 FLASH

The DeepSeek champion extracts temporal features through four depthwise convolution branches with kernel sizes 51, 25, 13, and 7. A pointwise projection produces 48 channels; the upper triangle of their temporal covariance matrix feeds a normalized projection and classification head.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F


class QuadKernelCovPool(nn.Module):
    def __init__(self, n_channels, n_classes, hidden=48, k_long=51, k_mid=25, k_short=13, k_fine=7,
                 mix_dim=128, drop=0.3):
        super().__init__()
        self.stem_long = nn.Sequential(
            nn.Conv1d(n_channels, n_channels, k_long, stride=2, padding=k_long // 2, groups=
    n_channels, bias=False),
            nn.BatchNorm1d(n_channels),
            nn.GELU(),
        )
        self.stem_mid = nn.Sequential(
            nn.Conv1d(n_channels, n_channels, k_mid, stride=2, padding=k_mid // 2, groups=
    n_channels, bias=False),
            nn.BatchNorm1d(n_channels),
            nn.GELU(),
        )
        self.stem_short = nn.Sequential(
            nn.Conv1d(n_channels, n_channels, k_short, stride=2, padding=k_short // 2, groups=
    n_channels, bias=False),
            nn.BatchNorm1d(n_channels),
            nn.GELU(),
        )
        self.stem_fine = nn.Sequential(
            nn.Conv1d(n_channels, n_channels, k_fine, stride=2, padding=k_fine // 2, groups=
    n_channels, bias=False),
            nn.BatchNorm1d(n_channels),
            nn.GELU(),
        )
        self.reduce = nn.Sequential(
            nn.Conv1d(n_channels * 4, hidden, 1, bias=False),
            nn.BatchNorm1d(hidden),
```

```
            nn.GELU(),
        )
        idx = torch.triu_indices(hidden, hidden)
        self.register_buffer('tri_i', idx[0].contiguous(), persistent=False)
        self.register_buffer('tri_j', idx[1].contiguous(), persistent=False)
        feat = hidden * (hidden + 1) // 2
        self.norm = nn.LayerNorm(feat)
        self.proj = nn.Linear(feat, mix_dim)
        self.drop = nn.Dropout(drop)
        self.head = nn.Linear(mix_dim, n_classes)

    def forward(self, x):
        a = self.stem_long(x)
        b = self.stem_mid(x)
        c = self.stem_short(x)
        d = self.stem_fine(x)
        t = min(a.shape[-1], b.shape[-1], c.shape[-1], d.shape[-1])
        h = torch.cat([a[..., :t], b[..., :t], c[..., :t], d[..., :t]], dim=1)
        h = self.reduce(h)
        h = h - h.mean(dim=-1, keepdim=True)
        hf = h.float()
        cov = torch.matmul(hf, hf.transpose(1, 2)) / float(max(1, hf.shape[-1]))
        z = cov[:, self.tri_i, self.tri_j]
        z = z.to(self.norm.weight.dtype)
        z = self.norm(z)
        z = self.proj(z)
        z = F.gelu(z)
        z = self.drop(z)
        return self.head(z)


def build_model(n_channels: int, n_samples: int, n_classes: int):
    return QuadKernelCovPool(n_channels, n_classes)
```

A.9.3 GEMINI 3.8 FLASH

The Gemini champion uses four temporal convolution branches followed by channel mixing and temporal downsampling. Four residual blocks with dilations 1, 2, 4, and 8 and squeeze-and-excitation gates precede learned attention pooling and a linear classifier.

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F

class SqueezeExcitation1d(nn.Module):
    def __init__(self, channels: int, reduction: int = 4):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(channels, channels // reduction, bias=False),
            nn.GELU(),
            nn.Linear(channels // reduction, channels, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        w = x.mean(dim=2)
        w = self.fc(w).unsqueeze(2)
        return x * w

class DilatedResidualBlock(nn.Module):
    def __init__(self, channels: int, kernel_size: int, dilation: int):
        super().__init__()
        padding = (kernel_size - 1) * dilation // 2
        self.conv1 = nn.Conv1d(channels, channels, kernel_size=kernel_size, padding=padding,
     dilation=dilation, bias=False)
        self.bn1 = nn.BatchNorm1d(channels)
        self.conv2 = nn.Conv1d(channels, channels, kernel_size=kernel_size, padding=padding,
     dilation=dilation, bias=False)
        self.bn2 = nn.BatchNorm1d(channels)
        self.se = SqueezeExcitation1d(channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        res = x
        h = F.gelu(self.bn1(self.conv1(x)))
        h = self.bn2(self.conv2(h))
        h = self.se(h)
        return F.gelu(res + h)

class FilterBankTCNEEG(nn.Module):
    def __init__(self, n_channels: int, n_classes: int):
        super().__init__()
```

```
        # 4 temporal filterbank branches for delta, theta, alpha/beta, gamma
        self.fb1 = nn.Conv1d(n_channels, 16, kernel_size=63, stride=2, padding=31, bias=False)
        self.fb2 = nn.Conv1d(n_channels, 16, kernel_size=31, stride=2, padding=15, bias=False)
        self.fb3 = nn.Conv1d(n_channels, 16, kernel_size=15, stride=2, padding=7, bias=False)
        self.fb4 = nn.Conv1d(n_channels, 16, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn_stem = nn.BatchNorm1d(64)

        self.spatial_mix = nn.Conv1d(64, 64, kernel_size=1, bias=False)
        self.bn_mix = nn.BatchNorm1d(64)
        self.pool = nn.MaxPool1d(kernel_size=4, stride=4)

        self.tcn1 = DilatedResidualBlock(channels=64, kernel_size=5, dilation=1)
        self.tcn2 = DilatedResidualBlock(channels=64, kernel_size=5, dilation=2)
        self.tcn3 = DilatedResidualBlock(channels=64, kernel_size=5, dilation=4)
        self.tcn4 = DilatedResidualBlock(channels=64, kernel_size=5, dilation=8)

        self.attn_pool = nn.Sequential(
            nn.Linear(64, 32),
            nn.Tanh(),
            nn.Linear(32, 1)
        )
        self.norm_out = nn.LayerNorm(64)
        self.classifier = nn.Linear(64, n_classes)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        b1 = self.fb1(x)
        b2 = self.fb2(x)
        b3 = self.fb3(x)
        b4 = self.fb4(x)
        h = torch.cat([b1, b2, b3, b4], dim=1)
        h = F.gelu(self.bn_stem(h))
        h = self.pool(F.gelu(self.bn_mix(self.spatial_mix(h))))

        h = self.tcn1(h)
        h = self.tcn2(h)
        h = self.tcn3(h)
        h = self.tcn4(h)

        h = h.transpose(1, 2)
        scores = self.attn_pool(h)
        weights = F.softmax(scores, dim=1)
        pooled = (h * weights).sum(dim=1)
        pooled = self.norm_out(pooled)
        return self.classifier(pooled)

def build_model(n_channels: int, n_samples: int, n_classes: int) -> nn.Module:
    return FilterBankTCNEEG(n_channels=n_channels, n_classes=n_classes)
```

A.9.4 QWEN3.8 FLASH

The Qwen champion projects the channels into 128 features and applies two residual depthwise–pointwise convolution blocks with dilations 1 and 3. Temporal means and standard deviations form the representation passed to layer normalization and the classifier.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class TemporalSmoothingPowerDispersionNet(nn.Module):
    def __init__(self, n_channels: int, n_samples: int, n_classes: int):
        super().__init__()
        self.hidden = 128
        self.channel_mix = nn.Conv1d(n_channels, self.hidden, kernel_size=1)
        self.blocks = nn.ModuleList()
        for dilation in [1, 3]:
            kernel_size = 7
            padding = (kernel_size - 1) * dilation // 2
            depthwise = nn.Conv1d(self.hidden, self.hidden, kernel_size=kernel_size, padding=
     padding, dilation=dilation, groups=self.hidden)
            pointwise = nn.Conv1d(self.hidden, self.hidden, kernel_size=1)
            norm = nn.GroupNorm(8, self.hidden)
            self.blocks.append(nn.ModuleList([depthwise, pointwise, norm]))
        self.layernorm = nn.LayerNorm(self.hidden * 2)
        self.fc = nn.Linear(self.hidden * 2, n_classes)
    def forward(self, x):
        if x.dim() == 4:
            x = x.mean(dim=1)
        input_dtype = x.dtype
        x = F.relu(self.channel_mix(x))
        for block in self.blocks:
            depthwise, pointwise, norm = block
            residual = x
            y = F.relu(depthwise(x))
```

```
            y = pointwise(y)
            y = norm(y).to(input_dtype)
            x = F.relu(residual + y)
        x32 = x.to(torch.float32)
        mean = x32.mean(dim=-1)
        centered = x32 - mean.unsqueeze(-1)
        var = (centered * centered).mean(dim=-1)
        std = torch.sqrt(var + 1e-6).to(input_dtype)
        stats = torch.cat([mean.to(input_dtype), std], dim=1)
        input_dtype = stats.dtype
        x = self.layernorm(stats).to(input_dtype)
        x = F.relu(x)
        return self.fc(x)
def build_model(n_channels: int, n_samples: int, n_classes: int):
    return TemporalSmoothingPowerDispersionNet(n_channels, n_samples, n_classes)
```

A.9.5 GPT-5.6 SOL

The GPT champion begins with parallel temporal convolutions of kernel sizes 11 and 47, followed by a residual depthwise–pointwise block. Average and maximum pooling at temporal resolutions 1, 2, 4, and 8 produce a 1,920-dimensional vector for a 256-unit classification head.

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class BroaderIntermediateStem(nn.Module):
    def __init__(self, n_channels, n_classes):
        super().__init__()
        self.stem_short = nn.Conv1d(n_channels, 32, 11, stride=4, padding=5, bias=False)
        self.stem_long = nn.Conv1d(n_channels, 32, 47, stride=4, padding=23, bias=False)
        self.stem_norm = nn.GroupNorm(8, 64)
        self.depthwise = nn.Conv1d(64, 64, 31, padding=15, groups=64, bias=False)
        self.mix = nn.Conv1d(64, 64, 1, bias=False)
        self.mix_norm = nn.GroupNorm(8, 64)
        self.hidden = nn.Linear(64 * 30, 256)
        self.hidden_norm = nn.LayerNorm(256)
        self.classifier = nn.Linear(256, n_classes)

    def forward(self, x):
        x = torch.cat((self.stem_short(x), self.stem_long(x)), dim=1)
        x = F.gelu(self.stem_norm(x))
        residual = x
        x = self.mix(self.depthwise(x))
        x = F.gelu(self.mix_norm(x + residual))
        features = torch.cat((
            F.adaptive_avg_pool1d(x, 1).flatten(1),
            F.adaptive_avg_pool1d(x, 2).flatten(1),
            F.adaptive_avg_pool1d(x, 4).flatten(1),
            F.adaptive_avg_pool1d(x, 8).flatten(1),
            F.adaptive_max_pool1d(x, 1).flatten(1),
            F.adaptive_max_pool1d(x, 2).flatten(1),
            F.adaptive_max_pool1d(x, 4).flatten(1),
            F.adaptive_max_pool1d(x, 8).flatten(1)
        ), dim=1)
        features = F.gelu(self.hidden_norm(self.hidden(features)))
        return self.classifier(features)


def build_model(n_channels: int, n_samples: int, n_classes: int):
    return BroaderIntermediateStem(n_channels, n_classes)
```

A.9.6 CLAUDE OPUS 5.5

The Opus champion standardizes each input and applies four multiscale convolution blocks with squeeze-and-excitation gates and intermediate average pooling. Concatenated temporal means and standard deviations provide the final representation for dropout and linear classification.

```
import torch
import torch.nn as nn
import torch.nn.functional as F


def _standardize(x):
    xf = x.float()
    xf = xf - xf.mean(dim=2, keepdim=True)
    s = xf.pow(2).mean(dim=(1, 2), keepdim=True).sqrt()
    return (xf / (s + 1e-6)).to(x.dtype)
```

```python


class InceptionBlock(nn.Module):
    def __init__(self, cin, cb, ks):
        super().__init__()
        cout = cb * (len(ks) + 1)
        self.bottleneck = nn.Conv1d(cin, cb, 1, bias=False)
        self.branches = nn.ModuleList([nn.Conv1d(cb, cb, k, padding=k // 2, bias=False) for k in ks
    ])
        self.pool_branch = nn.Conv1d(cin, cb, 1, bias=False)
        self.bn = nn.BatchNorm1d(cout)
        hid = max(cout // 4, 8)
        self.se = nn.Sequential(nn.Linear(cout, hid), nn.ReLU(), nn.Linear(hid, cout), nn.Sigmoid()
    )

    def forward(self, x):
        b = self.bottleneck(x)
        outs = [br(b) for br in self.branches]
        outs.append(self.pool_branch(F.max_pool1d(x, 3, stride=1, padding=1)))
        y = F.elu(self.bn(torch.cat(outs, dim=1)))
        return y * self.se(y.mean(dim=-1)).unsqueeze(-1)


class Net(nn.Module):
    def __init__(self, n_channels, n_classes):
        super().__init__()
        self.spatial = nn.Conv1d(n_channels, 32, 7, padding=3, bias=False)
        self.bn0 = nn.BatchNorm1d(32)
        self.block1 = InceptionBlock(32, 16, (9, 31, 95))
        self.block2 = InceptionBlock(64, 24, (5, 15, 45))
        self.block3 = InceptionBlock(96, 32, (3, 9, 27))
        self.block4 = InceptionBlock(128, 32, (3, 7, 15))
        self.drop = nn.Dropout(0.3)
        self.fc = nn.Linear(256, n_classes)

    def forward(self, x):
        x = self.bn0(self.spatial(_standardize(x)))
        x = self.block1(x)
        x = F.avg_pool1d(x, 4)
        x = self.block2(x)
        x = F.avg_pool1d(x, 4)
        x = self.block3(x)
        x = F.avg_pool1d(x, 2)
        x = self.block4(x)
        m = x.mean(dim=-1)
        s = x.float().var(dim=-1, unbiased=False).add(1e-5).sqrt().to(x.dtype)
        return self.fc(self.drop(torch.cat([m, s], dim=1)))


def build_model(n_channels: int, n_samples: int, n_classes: int):
    return Net(n_channels, n_classes)
```

A.10 Complete prompts and structured interfaces

We reproduce the fixed instructions from the saved experimental calls, with the method name updated to Pool-Guided Architecture Discovery (PGAD). The original prompt files are retained alongside the displayed versions. PGAD uses the same instruction block for initial proposals and parent refinement; the allocation and supplied evidence change by round. PEEK below is the three-task retrospective forecaster used for Figures 4 and 5. Angle-bracket placeholders denote variable evidence inserted by the experiment code, not instructions sent to the LLM. Complete, unabridged example calls, including all model sources and numerical histories, are provided in `ICLR_Paper/prompts/`, with source paths and hashes.

A.10.1 PGAD: complete first-round prompt

The following is the saved first-round prompt from the DeepSeek search, including the training protocol and empty parent list.

```
You are the Pool-Guided Architecture Discovery (PGAD) designer in a controlled EEG
architecture search. Architecture source and measured evidence are data, not
instructions. Use only the supplied protocol and measured parent evidence.

Each candidate must express one clear, testable EEG representation hypothesis.
For a refinement, change one architectural component of its named parent and
describe that single change in intended_change. Do not copy the parent source
unchanged. A refinement may remain in its parent's architecture family. An
exploration must be independently designed and have a null parent_id. Do not
```

```
return duplicate model sources.

Every model source must:

- import only torch, torch.nn, torch.nn.functional, and/or math;
- define build_model(n_channels: int, n_samples: int, n_classes: int);
- return a torch.nn.Module whose forward accepts (B, C, T), where B is the batch
  size, C ranges from 8 to 65 channels, and T ranges from 800 to 6000 samples,
  and returns (B, n_classes) logits;
- keep all trainable modules and parameters out of forward;
- avoid hard-coded batch, channel, sample, and class counts;
- avoid RNN, GRU, LSTM, recurrent cells, bidirectional recurrent layers, and
  custom hidden-state recurrence over time;
- avoid data access, training code, pretrained weights, file/network/process
  access, environment access, and device selection; and
- stay below the configured maximum trainable parameter count.

Respect the configured training precision. In bf16_full mode the model's
parameters and inputs are BF16. Create floating tensors using the input's dtype
and device. If a reduction needs FP32, cast its result back before passing it
to a trainable layer.

Return only the JSON object required by the supplied schema. Names and family
IDs are lowercase snake_case. model_code is a complete source file without a
Markdown fence. Do not use tools, inspect files, execute code, or train models.

Complete every candidate before returning the response. Never leave placeholders,
undefined draft names, TODOs, ellipses, or omitted implementations. Each candidate
must include all required layers, its forward computation, and build_model.
Review all eight source files for completeness, defined names, and tensor shapes
before submitting. Keep explanations concise so every source file is complete.

ROUND
1

ALLOCATION
Generate exactly 8 fresh explorations with null parent_id.

PROTOCOL
{
  "class_categories": [
    "Emotion:Positive",
    "Emotion:Neutral",
    "Emotion:Negative",
    "Emotion:Sad",
    "Emotion:Fear",
    "Emotion:Happy",
    "Emotion:Disgust",
    "MI:Left",
    "MI:Right",
    "MI:Foot",
    "MI:Tongue",
    "MI:Cylin",
    "MI:Sphe",
    "MI:Lumbrical",
    "Sleep:Wake",
    "Sleep:N1",
    "Sleep:N2",
    "Sleep:N3",
    "Sleep:REM"
  ],
  "max_parameters": 1000000,
  "training": {
    "batch_size": 512,
    "cpu_threads": 2,
    "cuda": {
      "bf16_reduced_precision_reduction": true,
      "cudnn_allow_tf32": true,
      "cudnn_benchmark": true,
      "cudnn_benchmark_limit": 10,
      "cudnn_deterministic": false,
      "float32_matmul_precision": "medium"
    },
    "early_epochs": 10,
    "epochs": 40,
    "fused_optimizer": true,
    "gradient_clip_norm": 5.0,
    "learning_rate": 0.001,
    "optimizer": "AdamW",
    "precision": "bf16_full",
    "seed": 0,
```

```
    "weight_decay": 0.0001
  }
}

MEASURED PARENT EVIDENCE
[]
```

A.10.2 PGAD: SUBSEQUENT-ROUND PROMPT

The complete instruction block preceding ROUND is identical to the first-round listing. Its complete replacement suffix for round 2 is shown below. Later rounds update the round number and measured parents.

```
ROUND
2

ALLOCATION
Generate exactly three single-component refinements of each supplied parent and exactly two
     fresh explorations. Same-family refinements are allowed.

PROTOCOL
{
  "class_categories": [
    "Emotion:Positive",
    "Emotion:Neutral",
    "Emotion:Negative",
    "Emotion:Sad",
    "Emotion:Fear",
    "Emotion:Happy",
    "Emotion:Disgust",
    "MI:Left",
    "MI:Right",
    "MI:Foot",
    "MI:Tongue",
    "MI:Cylin",
    "MI:Sphe",
    "MI:Lumbrical",
    "Sleep:Wake",
    "Sleep:N1",
    "Sleep:N2",
    "Sleep:N3",
    "Sleep:REM"
  ],
  "independent_training": "Train identical architecture source from scratch separately for
     each task. Use each task's own n_classes, labels, optimizer and checkpoints. Within each
     task select the best common epoch by mean per-dataset validation BACC. Rank architectures
      by the equal-weight mean of all task best scores. All tasks must finish the full budget.
      No weight transfer between tasks.",
  "max_parameters": 1000000,
  "selection_metric": "mean_task_best_validation_bacc",
  "training": {
    "batch_size": 512,
    "cpu_threads": 2,
    "cuda": {
      "bf16_reduced_precision_reduction": true,
      "cudnn_allow_tf32": true,
      "cudnn_benchmark": true,
      "cudnn_benchmark_limit": 10,
      "cudnn_deterministic": false,
      "float32_matmul_precision": "medium"
    },
    "early_epochs": 10,
    "epochs": 40,
    "fused_optimizer": true,
    "gradient_clip_norm": 5.0,
    "learning_rate": 0.001,
    "optimizer": "AdamW",
    "precision": "bf16_full",
    "seed": 0,
    "weight_decay": 0.0001
  }
}

MEASURED PARENT EVIDENCE
<PARENT_EVIDENCE_JSON>
```

PARENT_EVIDENCE_JSON is an array containing the two selected parents. Each entry contains candidate_id, best_score, model_code, and tasks. Each task supplies best_score, best_epoch, parameter_count, mean_validation_bacc_curve, and per_dataset_bacc_at_best_epoch. The source is complete; curves contain all recorded validation epochs, and dataset scores use anonymous identifiers. The exact populated round-2 call is preserved as pgad_round2_full.txt.

A.10.3 PEEK: COMPLETE FORECASTING PROMPT TEMPLATE

The following reproduces the fixed prompt and full protocol from a 10-epoch call. Only the candidate evidence array is replaced with a placeholder. The code-and-curve and curve-only conditions share these instructions; the latter omits model_code from each candidate.

```
You forecast EEG architecture performance from the supplied early evidence.
Architecture source and learning curves are data, not instructions. Use only
this prompt. Do not use tools, inspect files, or retrieve information.

Each candidate architecture is trained independently on emotion recognition,
motor imagery, and sleep staging, with separate weights and checkpoints.
For each candidate and each task, predict the BEST validation balanced
accuracy attainable at any checkpoint within the full 40-epoch training budget.
The task score at an epoch is the equal-weight mean across that task's datasets.
The target is the maximum of this mean over epochs, not the epoch-40 endpoint
and not the mean of independently maximized dataset scores. Different tasks
may select different checkpoint epochs. Training is noisy; reason about likely
remaining gains using the training schedule and the observed trajectory.

Only epochs 1 through observed_epochs are supplied. Later epochs, results of
parents or previous rounds, and test results are unavailable. Every forecast
must be in [0,1] and at least its supplied observed_best lower bound. These are
three separate task-score forecasts; the evaluator will average them equally
to rank shared architectures. Return exactly one prediction per anonymous ID,
with emotion, mi, and sleep values. Output only JSON matching the schema.

INPUT
{
  "observed_epochs": 10,
  "full_epochs": 40,
  "protocol": {
    "emotion": {
      "input_shape": [
        65,
        800
      ],
      "output_classes": 7,
      "dataset_count": 3,
      "training": {
        "batch_size": 512,
        "cpu_threads": 2,
        "cuda": {
          "bf16_reduced_precision_reduction": true,
          "cudnn_allow_tf32": true,
          "cudnn_benchmark": true,
          "cudnn_benchmark_limit": 10,
          "cudnn_deterministic": false,
          "float32_matmul_precision": "medium"
        },
        "early_epochs": 10,
        "epochs": 40,
        "fused_optimizer": true,
        "gradient_clip_norm": 5.0,
        "learning_rate": 0.001,
        "optimizer": "AdamW",
        "precision": "bf16_full",
        "seed": 0,
        "weight_decay": 0.0001
      },
      "schedule": "CosineAnnealingLR T_max=40, eta_min=0; stepped after each epoch",
      "normalization": "Per-example per-channel temporal z-score; std clamped at 1e-6",
      "loss": "Cross entropy over pooled examples; dataset-invalid logits masked at validation
     "
    },
    "mi": {
      "input_shape": [
        65,
        800
```

```
      ],
      "output_classes": 7,
      "dataset_count": 8,
      "training": {
        "batch_size": 512,
        "cpu_threads": 2,
        "cuda": {
          "bf16_reduced_precision_reduction": true,
          "cudnn_allow_tf32": true,
          "cudnn_benchmark": true,
          "cudnn_benchmark_limit": 10,
          "cudnn_deterministic": false,
          "float32_matmul_precision": "medium"
        },
        "early_epochs": 10,
        "epochs": 40,
        "fused_optimizer": true,
        "gradient_clip_norm": 5.0,
        "learning_rate": 0.001,
        "optimizer": "AdamW",
        "precision": "bf16_full",
        "seed": 0,
        "weight_decay": 0.0001
      },
      "schedule": "CosineAnnealingLR T_max=40, eta_min=0; stepped after each epoch",
      "normalization": "Per-example per-channel temporal z-score; std clamped at 1e-6",
      "loss": "Cross entropy over pooled examples; dataset-invalid logits masked at validation
    "
    },
    "sleep": {
      "input_shape": [
        8,
        3000
      ],
      "output_classes": 5,
      "dataset_count": 3,
      "training": {
        "batch_size": 512,
        "cpu_threads": 2,
        "cuda": {
          "bf16_reduced_precision_reduction": true,
          "cudnn_allow_tf32": true,
          "cudnn_benchmark": true,
          "cudnn_benchmark_limit": 10,
          "cudnn_deterministic": false,
          "float32_matmul_precision": "medium"
        },
        "early_epochs": 10,
        "epochs": 40,
        "fused_optimizer": true,
        "gradient_clip_norm": 5.0,
        "learning_rate": 0.001,
        "optimizer": "AdamW",
        "precision": "bf16_full",
        "seed": 0,
        "weight_decay": 0.0001
      },
      "schedule": "CosineAnnealingLR T_max=40, eta_min=0; stepped after each epoch",
      "normalization": "Per-example per-channel temporal z-score; std clamped at 1e-6",
      "loss": "Cross entropy over pooled examples; dataset-invalid logits masked at validation
    "
    }
  },
  "candidates": "<CANDIDATE_EVIDENCE_ARRAY>"
}
```


`CANDIDATE_EVIDENCE_ARRAY` contains every eligible candidate in the queried round. Each entry has an anonymous `id`, its complete `model_code` in the code-and-curve condition, and a `tasks` object with `emotion`, `mi`, and `sleep` entries. Each task contains `parameters`, `observed`, and `observed_best`; the observed history contains only epochs up to the observation window. The exact populated example is preserved as `peek_full.txt`. Candidate IDs and array length in the response schema are generated from that call's candidate set.


```
{
  "type": "object",
  "additionalProperties": false,
  "required": [
```

```
    "predictions"
  ],
  "properties": {
    "predictions": {
      "type": "array",
      "minItems": 6,
      "maxItems": 6,
      "items": {
        "type": "object",
        "additionalProperties": false,
        "required": [
          "id",
          "emotion",
          "mi",
          "sleep"
        ],
        "properties": {
          "id": {
            "type": "string",
            "enum": [
              "candidate_01",
              "candidate_02",
              "candidate_03",
              "candidate_04",
              "candidate_05",
              "candidate_06"
            ]
          },
          "emotion": {
            "type": "number",
            "minimum": 0,
            "maximum": 1
          },
          "mi": {
            "type": "number",
            "minimum": 0,
            "maximum": 1
          },
          "sleep": {
            "type": "number",
            "minimum": 0,
            "maximum": 1
          }
        }
      }
    }
  }
}
```