Below is a clean, end-to-end implementation plan for building the UELM-4 system—the single-potential, Mirror-CDE, Dual-Simplex-State model with Continuous-Memory Measure (CMM), Wasserstein-Mirror Flow (WMF), symplectic-dissipative splitting, cache-as-constraint (CaC), tied readout, and a learned meta-solver. It's structured as a practical blueprint: repo layout → modules → interfaces → algorithms → tests → training → deployment.

---

0) High-level build goals

- Usable LLM: throughput close to a small Transformer while offering long-context and memory efficiency.
- Single potential: one scalar objective $\Psi(P)$ with equilibrium solution $P^\star$.
- Minimal state: keep simplex state P (sparse, top-k), its image $Y=M^\top P$, and compact caches.
- Predictable latency: few iterations per token ($T\in\{1,2,3\}$) with early-exit guards.
- Modular: each innovation is a drop-in (you can ship a KL-only prox before WMF, table memory before CMM, etc.).

---

1) Repository structure

```
uelm4/
  README.md
  pyproject.toml
  uelm4/
    __init__.py
    config/
      defaults.yaml
      small.yaml
      base16k.yaml
    core/
      types.py              # Typed dataclasses for states, configs
      control_path.py           # Piecewise-linear path builder
      banded_ops.py             # Causal banded conv/FFT kernels
      preconditioners.py        # Woodbury, diagonal, low-rank
      optimizers.py           # Anderson, CG helpers (JVP-safe)
```

```
    solver_pdhg.py              # Mirror-PDHG/ADMM driver (Y,P,Λ)
    wmf_prox.py                 # Wasserstein-Mirror proximal updates
    kl_prox.py                  # KL-only (masked softmax) prox
    symp_diss_field.py          # Symplectic-dissipative vector field
    energy.py                   # Psi(P) construction, grads/JVPs
    implicit_autograd.py        # Custom backward for equilibria
  memory/
    cmm.py                      # Continuous Memory Measure (generator)
    landmarks.py                # Nyström dictionary, landmark ops
    shortlist.py                # ANN shortlist + support freezing
    rag_bridge.py               # Optional RAG injection into memory
    readout_tied.py             # Tied readout to memory (lex subset)
  model/
    embeddings.py               # Token, pos embeddings (causal)
    scout.py                    # Amortized initializer (meta)
    controller.py               # Meta-solver controller (β,τ,steps)
    uelm4_model.py              # Full UELM4 nn.Module
    decode.py                   # Autoregressive decode loop
  data/
    tokenization.py             # BPE/SentencePiece wrappers
    dataloaders.py              # Streaming dataloaders, packing
  train/
    train.py                    # Main training script
    losses.py                   # LM CE + auxiliary losses
    schedules.py                # β, τ, iteration schedules
    eval.py                     # PPL, long-context, calibration
    metrics.py                  # Energy, gaps, convergence metrics
  tests/
    test_energy.py
    test_solver.py
    test_memory.py
    test_wmf.py
    test_cde.py
    test_decode.py
scripts/
  build_landmarks.py            # Precompute/fit Nyström dictionary
  index_ann.py                  # Build ANN (IVF-PQ) over landmarks
  profile_decode.py             # Latency profiling
docs/
  design.md                     # Math derivations, invariants
  api.md                        # Public interfaces
  experiments.md                # Reproducible experiment sheets
```

---

2) Environment & dependencies

- Framework: PyTorch ≥ 2.3 (SDPA kernels, AMP), functorch for JVPs if needed.
- ANN: FAISS-GPU (IVF-PQ) or ScaNN; CPU fallback for dev.
- FFT: torch.fft; ensure cuFFT available.
- Config: OmegaConf / hydra; dataclasses for strong typing.
- Logging: Weights & Biases / TensorBoard.
- Determinism: set seeds, cudnn deterministic paths where feasible.

---

3) Core mathematical objects & invariants

3.1 State & shapes

- Sequence length n, width d, shortlist k.
- P: (n, k) simplex per token (masked where pattern not in shortlist).
- Y: (n, d) = $M^\top P$.
- M: memory atoms: either table (K, d) (phase 1) or CMM landmarks (K0, d) + generator g_ϕ (phase 2).
- Λ: (n, d) dual for constraint $Y=M^\top P$.

3.2 Key invariants

- Simplex: P[i].sum() == 1, P[i] ≥ 0.
- Causality: no operator uses future tokens; banded kernels strictly lower-triangular.
- Energy descent: per (outer) iteration, Ψ non-increasing up to tolerance.
- Cache coherence: CaC penalty only anchors past slices.

---

4) Implementation phases (ship incremental value)

Phase A (MVP: KL-prox + table memory)

1. Tokenization, embeddings, control path, causal banded CDE with symp-diss split.
2. Table memory M, shortlist (FAISS) per token, KL-prox (masked softmax).
3. Two-block solver (Y-step + P-step) with Anderson/CG; CaC.
4. Tied readout to M lex subset (or to embeddings), Scout initializer.
5. Train on small corpus; measure convergence and latency.

Phase B (Full UELM-4)

6. Replace table with CMM (generator + landmarks); shortlist from CMM.

7. WMF prox (Sinkhorn-style on shortlist) with inertia τ; meta-solver controller.

8. Implicit differentiation custom autograd for equilibrium solves.

9. Quantization & Nyström compression pipelines; long-context benchmarks.

Phase C (Extensions)

10. RAG bridge; reversible share tuning; coarse-to-fine CDE; distributional readout from P.

---

5) Module contracts (interfaces)

5.1 Control path

```
# uelm4/core/control_path.py
@dataclass
class Path:
   knots: torch.Tensor  # (n, dx)
   times: torch.Tensor  # (n,)

def make_control_path(E: torch.Tensor, method: str="piecewise_linear") -> Path:
   """Builds piecewise-linear path X from embeddings E (n,d) -> (n,dx)."""
```

5.2 Banded ops & vector field (symplectic–dissipative)

```
# uelm4/core/symp_diss_field.py
class BandedField(nn.Module):
    def __init__(self, d: int, band: int, spectral_norm: bool=True):
        ...

    def forward(self, Y: torch.Tensor, t_feat: torch.Tensor) -> tuple[Tensor, Tensor]:
        """Returns (S(Y,t), D(Y,t)), where S approx skew-symmetric, D PSD."""

def cde_split_step(Y, X: Path, field: BandedField, precond) -> torch.Tensor:
    """One Strang-split step (symplectic half, dissipative, symplectic half)."""
```

5.3 Memory: CMM & landmarks

```
# uelm4/memory/cmm.py
class CMMemory(nn.Module):
    def __init__(self, d: int, K0: int, generator_cfg):
        self.landmarks = nn.Parameter(torch.randn(K0, d)*0.02)
        self.generator = GeneratorNet(generator_cfg)  # optional
    def sample_atoms(self, q: torch.Tensor, num: int) -> torch.Tensor: ...
    def landmarks_view(self) -> torch.Tensor: ...

# uelm4/memory/shortlist.py
def shortlist(
    E_prefix: torch.Tensor,
    memory: CMMemory | Tensor,    # support table first
    k: int, causal: bool, idx
) -> list[torch.Tensor]:
    """Returns per-token shortlist indices (length n; each shape (k,))."""
```

5.4 Prox steps (KL & WMF)

```
# uelm4/core/kl_prox.py
def kl_masked_softmax(P, scores, mask) -> torch.Tensor:
    """KL-prox: p_new ∝ p_old * exp(η * scores), masked to support."""

# uelm4/core/wmf_prox.py
def wmf_prox_step(p, scores, cost, tau, lam_kl, iters=3) -> torch.Tensor:
    """
    Wasserstein-Mirror prox on shortlist:
    - p: (k,)
    - scores: fit signal (k,)
    - cost: ground metric matrix (k,k)
    - tau: inertia
    - lam_kl: KL weight
```

```
    Returns p_new on the simplex.
    """
```

5.5 Solver (Mirror-PDHG/ADMM)

```
# uelm4/core/solver_pdhg.py
@dataclass
class SolverState:
    P: torch.Tensor      # (n,k)
    Y: torch.Tensor      # (n,d)
    Λ: torch.Tensor      # (n,d)
    Kset: torch.Tensor   # (n,k) shortlist indices
    energy: float

class MirrorPDHG(nn.Module):
    def __init__(self, cfg, field: BandedField, memory, controller):
        ...
    def step(self, state: SolverState, batch_ctx) -> SolverState:
        """
        Executes: Y-step (precond grad/CG), P-step (KL or WMF prox),
        Dual update, support freeze, early-exit on energy gap.
        """
```

5.6 Readout (tied)

```
# uelm4/memory/readout_tied.py
class TiedReadout(nn.Module):
    def __init__(self, M_lex: torch.Tensor, scale: float=1.0):
        ...
    def forward(self, Y: torch.Tensor) -> torch.Tensor:  # logits (n, vocab)
        return self.scale * (Y @ M_lex.T) + self.bias
```

5.7 Model & decode

```
# uelm4/model/uelm4_model.py
class UELM4(nn.Module):
    def __init__(self, cfg):
        self.embed = Embeddings(cfg)
        self.scout = Scout(cfg)
        self.memory = CMMemory(...) or torch.nn.Parameter(M)  # phase A
        self.field = BandedField(cfg.d, cfg.band)
        self.readout = TiedReadout(...)
        self.controller = MetaController(...)
        self.solver = MirrorPDHG(cfg.solver, self.field, self.memory, self.controller)

    def forward(self, tokens):
```

```
        # teacher-forcing: build path, shortlist, init P,Y with Scout
        # run T iterations -> logits
        ...

# uelm4/model/decode.py
def autoregressive_decode(model, prompt_ids, max_new_tokens, budget_cfg):
    # Warm start caches, lazy shortlist refresh, CaC anchoring
    # T in {1,2,3}, early-exit on energy gap; return generated ids
```

---

6) Algorithms & math in code terms

6.1 Energy & gradients

- Psi(P) in energy.py: compute CDE residual, entropy term, CaC penalty, causal reg, smoothness reg.
- Provide JVP/VJP utilities to support implicit differentiation and CG solves without materializing Jacobians.

6.2 Y-step (least squares in Y)

- Residual $R(Y) = Y - Y_0 - \int \tilde G(Y,t)\,dX_t$.
- One preconditioned gradient or a few CG iterations with:
  - Preconditioner: woodbury_precond built from top-k atom spans (preconditioners.py).
  - Stop when relative residual below y_tol.

6.3 P-step (probability update)

- Compute fit scores per token on shortlist: scores = M[Kset] @ ξ_i, where $\xi_i = \Lambda_i + \rho (Y_i - (M^\top P)_i)$.
- KL-prox (phase A) or WMF prox (phase B):
  - For WMF: build ground cost (k,k): squared distances of shortlisted atoms, precomputed per batch.

    - 2–3 scaling iterations; project to simplex; apply support hysteresis to reduce churn.

6.4 Dual update

- $\Lambda \leftarrow \Lambda + \rho (Y - M^T P)$; optionally clip duals to keep numerics stable.

6.5 Symplectic–dissipative split

- $S(Y,t)$ parameterized as low-rank skew part: $S = A - A^T$ with A banded; D as PSD via $B^T B$ with banded B.
- Strang step uses half-steps of S (cheap linear ops) and a full dissipative correction.

6.6 Cache-as-constraint (CaC)

- If previous Y_prev exists, add $\kappa \cdot \|\Pi_{past}(Y) - \mathrm{Advect}(Y_{prev})\|^2$.
- Advect can be identity or a 1-tap banded conv to shift time by one step.

6.7 Controller (meta-solver)

- Input features: entropy $H(P_i)$, local CDE residual norms, energy drop, shortlist dispersion.
- Outputs: β_{fac}, τ, Y-step size, preconditioner rank; trained by distillation from a higher-budget teacher (T_{hi} vs T_{lo} losses).

---

7) Testing strategy

7.1 Unit tests

- Energy: Psi(P) decreases after one full iteration (within tolerance).
- Simplex: P nonnegative, sums to 1; KL-prox/WMF prox preserve simplex.

- Causality: no future indices used (assert masks).
- WMF prox: transport mass conservation; converges to KL solution when tau→0.
- CDE split: reversible part preserves norm in absence of D and residual term (up to fp error).

7.2 Gradient checks

- Finite-diff gradients for small cases (n=8,k=8,d=16).
- Implicit backward correctness: compare to unrolled backprop for T=8 (small toy).

7.3 Integration tests

- Decode determinism with fixed seeds.
- Cache: tokens t→t+1 require ≤ previous iterations when CaC on.
- Shortlist recall: synthetic “needle” memory must land in top-k with >95% probability.

7.4 Performance tests

- ms/token under (T,k,w) grids; GPU memory footprint vs context length.

---

8) Data pipeline

- Tokenizer: BPE/SentencePiece; causal packing for efficient training.
- Datasets: streaming sharded (JSONL or mmap); curriculum on context (4k→32k→64k).
- Filtering: deduplication, quality score; optional code/math subsets for long-range.

---

9) Training plan

9.1 Losses

- LM CE (teacher forcing).
- Stationarity regularizer: $\|Y - M^T P\|^2$ (already in augmented Lagrangian).
- Energy penalty: small weight on $\|\nabla\Psi(P^*)\|^2$ early only.
- Amortization (Scout): MSE to P^* or Y^* (stop-grad target).
- Contrastive memory shaping (optional): InfoNCE for shortlist retrieval.

9.2 Schedules & curriculum

- Start with KL-prox only; introduce WMF after stability (epoch ~N).
- β: $0.5 \to 1.5$ over solver iterations; τ: $0.2 \to 0.05$.
- Increase band width w and context length as convergence stabilizes.
- Controller distillation: run a teacher solver at $T=4$ on a subset; train controller to match the $T=2$ outcome.

9.3 Optimizer & numerics

- AdamW, cosine LR; AMP; grad clip 1.0; spectral norm on banded kernels.
- Log-sum-exp everywhere (stabilize softmax).
- Clamp $\|m_k\|$ and dual norms.

---

10) Evaluation & ablations

- PPL on standard corpora; long-context recall (needle-in-haystack at 32k–128k).
- Latency vs (T,k,w); iteration histos per token.
- Calibration (ECE), entropy of P.

- Ablations: table vs CMM, KL vs WMF, CaC on/off, split on/off, tied readout on/off, controller on/off.
- Transformer limit: set $T=1$, $\mu \to 0$, $k=K$ (small), compare to 2–4 layer Transformer.

---

11) Deployment & runtime

- Export: TorchScript/ONNX for inference; separate ANN server or embedded FAISS index on GPU.
- Quantization: int8 for M/landmarks and readout; bf16 elsewhere.
- Serving: budget tuner picks (T,k,w) per request; early-exit at token level.
- RAG: append retrieved vectors as atoms with mass in CMM; no code path change.

---

12) Config templates (YAML)

```yaml
# uelm4/config/base16k.yaml
model:
  d: 2048
  band: 256
  vocab_size: 32000
  tied_readout: true
memory:
  type: "table"          # "cmm" for phase B
  K: 131072
  K0: 8192               # landmarks used when type=cmm
  shortlist_k: 32
solver:
  T_train: 2
  T_infer: 1
  rho: 1.0
  beta_start: 0.5
  beta_end: 1.5
  tau_start: 0.2
  tau_end: 0.05
  early_exit_tol: 1e-3
```

```yaml
  use_wmf: false         # enable in phase B
field:
  symp_share: 0.6
  spectral_norm: true
cac:
  kappa: 0.25
controller:
  enabled: false         # enable in phase B
training:
  lr: 2.0e-4
  batch_tokens: 2_000_000
  amp: true
  grad_clip: 1.0
```

---

13) Skeleton: forward & decode (phase A: KL‑prox + table)

```python
# uelm4/model/uelm4_model.py
class UELM4(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.embed = Embeddings(cfg)
        self.field = BandedField(cfg.model.d, cfg.model.band, spectral_norm=True)
        self.memory = nn.Parameter(torch.randn(cfg.memory.K, cfg.model.d) * 0.02)
        self.readout = TiedReadout(self.memory[:cfg.model.vocab_size])  # simple tie
        self.solver = MirrorPDHG(cfg.solver, self.field, self.memory, controller=None)

    def forward(self, tokens):
        E = self.embed(tokens)                          # (n,d)
        X = make_control_path(E)
        Kset = shortlist(E, self.memory, cfg.memory.shortlist_k, causal=True, idx=None)
        P0 = init_simplex_from_scout(E, Kset)          # uniform if no Scout
        Y0 = self.memory[Kset].transpose(-1,-2) @ P0    # M^T P
        state = SolverState(P=P0, Y=Y0, Λ=torch.zeros_like(Y0), Kset=Kset, energy=float('inf'))
        for _ in range(cfg.solver.T_train):
            state = self.solver.step(state, batch_ctx={'X': X})
            if early_exit(state.energy): break
        logits = self.readout(state.Y)
        return logits
```

---

14) Skeleton: solver step (Mirror-PDHG; KL first, WMF later)

```
# uelm4/core/solver_pdhg.py (simplified)
class MirrorPDHG(nn.Module):
    def step(self, st: SolverState, ctx):
        X = ctx['X']
        # Y-step: preconditioned grad towards CDE consistency
        Y = cde_split_step(st.Y, X, self.field, precond=woodbury_from(st.P, st.Kset, self.memory))
        # Dual signal
        R = Y - self.memory[st.Kset].transpose(-1,-2) @ st.P
        Xi = st.Λ + self.cfg.rho * R
        scores = batched_matvec(self.memory[st.Kset], Xi)      # (n,k)
        # P-step: prox
        if self.cfg.use_wmf:
            cost = pairwise_sqdist(self.memory[st.Kset])       # (n,k,k)
            P = wmf_prox_step(st.P, scores, cost, tau=self.ctrl.tau(), lam_kl=self.ctrl.lam_kl())
        else:
            P = kl_masked_softmax(st.P, scores, mask=st.Kset)
        # Support freeze after first iter (not shown)
        # Dual update
        Λ = st.Λ + self.cfg.rho * (Y - self.memory[st.Kset].transpose(-1,-2) @ P)
        energy = compute_energy(P, Y, Λ, self.cfg, X, ...)
        return SolverState(P=P, Y=Y, Λ=Λ, Kset=st.Kset, energy=energy)
```

---

15) VRAM & complexity planning

- Compute per iter:
    - Y-step: banded conv O(n\,w\,d) or FFT O(n\log n\,d).
    - Memory ops: O(n\,k\,d) for M^T P and batched matvec.
    - WMF: O(n\,k^2) for small k (≤64), 2–3 iterations.
- VRAM: store P (n×k), Y (n×d), shortlist indices (n×k), and banded buffers; no layer-wise activations or KV stacks.

---

16) Risk controls & fallbacks

- If WMF unstable: set tau→0 to recover KL-prox.
- If CaC causes drift: reduce kappa or switch off for a few steps.
- If solver oscillates: increase dissipative share or rho, cap β, reduce step size; enable Anderson with small depth (m=3).
- If shortlist churns: enable support hysteresis and longer lazy refresh.

---

17) Deliverables checklist

- Unit & integration tests pass (CI).
- Profiling: ms/token vs (T,k,w) table in experiments.md.
- Reproducible training config (base16k.yaml), seed logs, exact commit hash.
- Ablation results (phase A vs B).
- Documentation of public APIs (docs/api.md).
- Example scripts: profile_decode.py, build_landmarks.py, index_ann.py.

---

18) What to implement first (practical ordering)

1. Phase A core: embeddings, control path, banded field (KL-only), table memory + shortlist, KL-prox, Y-step, solver loop, tied readout, decode.
2. Testing + profiling: ensure energy descent, simplex invariants, causal masks, basic latency.
3. Phase B upgrades: CMM generator + landmarks + ANN; WMF prox; controller; implicit backward.
4. Polish: CaC, symplectic tuning, coarse-to-fine CDE, quantization, RAG bridge.

This plan is designed so you can cut a thin vertical slice (Phase A) that is runnable and measurable, then layer in the advanced pieces (Phase B) without refactoring the core abstractions. If you want, I can translate any section into full code files matching the repo layout above (e.g., solver_pdhg.py, wmf_prox.py, symp_diss_field.py) and include docstrings and unit tests inline.