

Universal Stochastic Predictor

Phase 3: Core Orchestration

Implementation Team

February 19, 2026

Contents

1	Phase 3: Core Orchestration Overview	2
1.1	Tag Information	2
1.2	Scope	2
1.3	Design Principles	2
2	Sinkhorn Module (core/sinkhorn.py)	3
2.1	Volatility-Coupled Regularization	3
2.2	Entropy-Regularized OT (Scan-Based)	3
3	Fusion Module (core/fusion.py)	5
3.1	JKO-Weighted Fusion	5
3.2	Simplex Sanitization	5
4	Core Public API	6
4.1	Compliance Checklist	6
5	Phase 3 Summary	7
5.1	Phase 4 Integration Note	7

Chapter 1

Phase 3: Core Orchestration Overview

1.1 Tag Information

- **Tag:** impl/v2.0.3
- **Commit:** cb119d9
- **Status:** Complete, audited, and verified

Phase 3 implements the physical orchestration layer in `stochastic_predictor/core/`. This layer fuses heterogeneous kernel outputs using Wasserstein gradient flow (JKO) and entropic optimal transport (Sinkhorn) with volatility-coupled regularization.

1.2 Scope

Phase 3 covers:

- **Sinkhorn Regularization:** Volatility-coupled entropic regularization for stable optimal transport
- **Wasserstein Fusion:** JKO-weighted fusion of kernel predictions and confidence scores
- **Simplex Sanitization:** Enforced simplex constraints for kernel weights
- **Core API:** Exported fusion and Sinkhorn utilities via `core/__init__.py`

1.3 Design Principles

- **Zero-Heuristics Policy:** All parameters injected via `PredictorConfig`
- **JAX-Native:** Stateless functions compatible with JIT/vmap
- **Determinism:** Bit-exact reproducibility under configured XLA settings
- **Volatility Coupling:** Dynamic regularization tied to EWMA variance

Chapter 2

Sinkhorn Module (core/sinkhorn.py)

2.1 Volatility-Coupled Regularization

The entropic regularization parameter adapts to local volatility according to the specification:

$$\varepsilon_t = \max(\varepsilon_{\min}, \varepsilon_0 \cdot (1 + \alpha \cdot \sigma_t))$$

where $\sigma_t = \sqrt{\text{EMA variance}}$ and α is the coupling coefficient.

```
1 def compute_sinkhorn_epsilon(ema_variance, config):
2     sigma_t = jnp.sqrt(jnp.maximum(ema_variance, config.numerical_epsilon))
3     epsilon_t = config.sinkhorn_epsilon_0 * (1.0 + config.sinkhorn_alpha * sigma_t)
4     return jnp.maximum(config.sinkhorn_epsilon_min, epsilon_t)
```

2.2 Entropy-Regularized OT (Scan-Based)

The Sinkhorn iterations are implemented with `jax.lax.scan` to ensure predictable XLA lowering and to support per-iteration volatility coupling. The iteration count is controlled by `config.sinkhorn_max_iter`.

```
1 def volatility_coupled_sinkhorn(source_weights, target_weights, cost_matrix, ema_variance
2     , config):
3     log_a = jnp.log(jnp.maximum(source_weights, config.numerical_epsilon))
4     log_b = jnp.log(jnp.maximum(target_weights, config.numerical_epsilon))
5     f0 = jnp.zeros_like(source_weights)
6     g0 = jnp.zeros_like(target_weights)
7
8     def sinkhorn_step(carry, _):
9         f, g = carry
10        eps = compute_sinkhorn_epsilon(ema_variance, config)
11        f = _smin(cost_matrix - g[None, :], eps) + log_a
12        g = _smin(cost_matrix.T - f[None, :], eps) + log_b
13        return (f, g), None
14
15    (f_final, g_final), _ = jax.lax.scan(
16        sinkhorn_step, (f0, g0), None, length=config.sinkhorn_max_iter
17    )
18
19    epsilon_final = compute_sinkhorn_epsilon(ema_variance, config)
20    transport = jnp.exp((f_final[:, None] + g_final[None, :] - cost_matrix) /
21        epsilon_final)
22    safe_transport = jnp.maximum(transport, config.numerical_epsilon)
23    entropy_term = jnp.sum(safe_transport * (jnp.log(safe_transport) - 1.0))
24    reg_ot_cost = jnp.sum(transport * cost_matrix) + epsilon_final * entropy_term
25    row_err = jnp.max(jnp.abs(jnp.sum(transport, axis=1) - source_weights))
26    col_err = jnp.max(jnp.abs(jnp.sum(transport, axis=0) - target_weights))
27    max_err = jnp.maximum(row_err, col_err)
```

```
26     converged = max_err <= config.validation_simplex_atol
27     return SinkhornResult(
28         transport_matrix=transport,
29         reg_ot_cost=reg_ot_cost,
30         converged=jnp.asarray(converged),
31         epsilon=jnp.asarray(epsilon_final),
32         max_err=jnp.asarray(max_err),
33     )
```

Chapter 3

Fusion Module (core/fusion.py)

3.1 JKO-Weighted Fusion

The fusion step normalizes kernel confidences into a simplex and performs a JKO proximal update on weights:

$$\rho_{k+1} = \rho_k + \tau(\hat{\rho} - \rho_k)$$

```
1 def fuse_kernel_outputs(kernel_outputs, current_weights, ema_variance, config):
2     predictions = jnp.array([ko.prediction for ko in kernel_outputs]).reshape(-1)
3     confidences = jnp.array([ko.confidence for ko in kernel_outputs]).reshape(-1)
4     target_weights = _normalize_confidences(confidences, config)
5
6     cost_matrix = compute_cost_matrix(predictions, config)
7     sinkhorn_result = volatility_coupled_sinkhorn(
8         source_weights=current_weights,
9         target_weights=target_weights,
10        cost_matrix=cost_matrix,
11        ema_variance=ema_variance,
12        config=config,
13    )
14
15    updated_weights = _jko_update_weights(current_weights, target_weights, config)
16    PredictionResult.validate_simplex(updated_weights, config.validation_simplex_atol)
17
18    fused_prediction = jnp.sum(updated_weights * predictions)
19    return FusionResult(
20        fused_prediction=fused_prediction,
21        updated_weights=updated_weights,
22        free_energy=sinkhorn_result.reg_ot_cost,
23        sinkhorn_converged=sinkhorn_result.converged,
24        sinkhorn_epsilon=sinkhorn_result.epsilon,
25        sinkhorn_transport=sinkhorn_result.transport_matrix,
26    )
```

3.2 Simplex Sanitization

The simplex constraint is validated using the injected tolerance:

```
1 PredictionResult.validate_simplex(updated_weights, config.validation_simplex_atol)
```

Chapter 4

Core Public API

```
1 from .fusion import FusionResult, fuse_kernel_outputs
2 from .sinkhorn import SinkhornResult, compute_sinkhorn_epsilon,
  volatility_coupled_sinkhorn
```

4.1 Compliance Checklist

- **Zero-Heuristics:** All parameters injected via config
- **Volatility Coupling:** Implemented per specification
- **Simplex Validation:** Config-driven tolerance enforced
- **JAX-Native:** Pure functions and stateless modules

Chapter 5

Phase 3 Summary

Phase 3 delivers a concrete orchestration layer for Wasserstein fusion and JKO weight updates. The core layer is now physically present and ready for integration with the prediction pipeline.

5.1 Phase 4 Integration Note

In Phase 4, the orchestration pipeline is extended with ingestion validation and IO gates. The `orchestrate_step()` function signature is updated to accept observation metadata (`ProcessState`, `now_ns`) and integrates the ingestion gate prior to kernel execution. See `Implementation_v2.0.4_IO.tex` for complete documentation.