Python Implementation Guide for Universal Stochastic Predictors

Adaptive Meta-Prediction Development Consortium

February 21, 2026

Contents

Chapter 1

Environment and Technology Stack

This guide translates the universal algorithmic specification into a high-performance Python production ecosystem.

1.1 Library Selection

1.1.1 JAX: Numerical Compute Engine

JAX 0.4.38 is the foundational layer. It provides:

- **XLA**: JIT compilation to optimized CPU/GPU/TPU kernels
- **Automatic Differentiation**: `jax.grad`, `jax.jacfwd`, `jax.hessian`
- **Vectorization**: `vmap` for parallelism without explicit loops
- **Composability**: `jit(vmap(grad(...)))`

Design decision: JAX is mandatory because JKO requires Wasserstein gradients and Branches B/C require backprop through differential equations.

1.1.2 Equinox 0.13.4: Neural Framework

Equinox is used for Branch B (DGM) and Branch C (Neural ODEs). It is selected for:

1. Pure functional design compatible with JAX transforms
2. Native Pytrees with no manual conversion
3. Minimal hidden state or metaclass machinery
4. Full differentiability of parameters
5. Parameter filtering via `eqx.filter`

1.1.3 Diffrax 0.7.2: ODE/SDE Solvers

Diffrax integrates Branch C (Neural SDE/ODE).

1. JAX-native, fully JIT-able
2. End-to-end differentiable via adjoint methods
3. Supports Ito/Stratonovich SDEs
4. Adaptive solvers (Heun, Tsit5, Dopri5)

1.1.4 Signax 0.2.1: Log-Signatures

Signax provides signature and log-signature computation for Branch D.

- GPU-native kernels
- Differentiable signatures
- Configurable truncation depth
- Stable log-domain computation

1.1.5 OTT-JAX 0.6.0: Differentiable Optimal Transport

OTT-JAX implements stabilized log-domain Sinkhorn for the JKO orchestrator.

1.1.6 Full Stack Summary

Component	Library	Version	Role
Compute core	JAX	0.4.38	XLA, AD, vmap, jit
Neural models	Equinox	0.13.4	DGM (Branch B)
ODE/SDE solvers	Diffrax	0.7.2	Branch C
Signatures	Signax	0.2.1	Branch D
Optimal Transport	OTT-JAX	0.6.0	JKO orchestrator
Wavelets	PyWavelets	1.9.0	WTMM (SIA)

1.2 Global Numerical Precision

Enable x64 globally when running Malliavin and signature computations:

```
import jax
jax.config.update('jax_enable_x64', True)
```

This improves stability for Holder exponent estimation, Skorokhod integrals, and Sinkhorn convergence at small ε.

1.3 Directory Architecture (Implementation Constraint)

All implementations must respect the five-layer layout:

- `stochastic_predictor/api/`: external contracts, config, validation
- `stochastic_predictor/core/`: orchestration, JKO flow
- `stochastic_predictor/kernels/`: pure XLA kernels (A, B, C, D)
- `stochastic_predictor/io/`: I/O, snapshots, sanitization
- `tests/`: causal and parity validation

1.4 Dependency Pinning (Golden Master)

Dynamic version ranges are prohibited. Environments must be frozen to exact versions. Use strict pinning in CI/CD and production.

Cross-Platform Support: JAX/JAXlib use PEP 508 environment markers in `requirements.txt` to enable platform-specific versions (macOS Intel/ARM, Linux, Windows) while maintaining single-file dependency specification. All other dependencies are platform-independent.

Chapter 2

Module 1: Identification Engine (SIA)

2.1 WTMM Estimation with Async Callback

Use `jax.pure_callback` to call PyWavelets on CPU without breaking JIT:

```
import pywt
import jax
import jax.numpy as jnp
import numpy as np

class WTMMEstimator:
    def __init__(self, n_scales=40, j_min=1.0, j_max=6.0):
        powers = jnp.linspace(j_min, j_max, num=n_scales)
        self.scales = jnp.power(2.0, powers)
        self.wavelet = 'gaus1'

    def compute_cwt_safe(self, signal):
        result_shape = (len(self.scales), signal.shape[0])

        def _cwt_cpu(s, sc):
            coefs, _ = pywt.cwt(np.array(s), np.array(sc), 'gaus1')
            return coefs.astype(np.float32)

        coefs = jax.pure_callback(
            _cwt_cpu,
            jax.ShapeDtypeStruct(result_shape, jnp.float32),
            signal, self.scales
        )
        return coefs
```

Chapter 3

Module 2: Prediction Kernels

3.1 Branch A: Levy Processes

```python
import jax.numpy as jnp
from jax import random

def simulate_stable_levy(key, alpha, beta, gamma, delta, n_samples):
    k1, k2 = random.split(key)
    phi = random.uniform(k1, shape=(n_samples,), minval=-jnp.pi/2, maxval=jnp.pi/2)
    w = random.exponential(k2, shape=(n_samples,))

    s_alpha_beta = (1 + (beta * jnp.tan(jnp.pi * alpha / 2))**2)**(1 / (2 * alpha))
    b_alpha_beta = jnp.arctan(beta * jnp.tan(jnp.pi * alpha / 2)) / alpha

    term1 = s_alpha_beta * (jnp.sin(alpha * (phi + b_alpha_beta))) / ((jnp.cos(phi))**(1/alpha))
    term2 = ((jnp.cos(phi - alpha * (phi + b_alpha_beta))) / w)**((1 - alpha) / alpha)

    z = term1 * term2
    return gamma * z + delta
```

3.2 Branch B: DGM-PDE Solvers

```python
import equinox as eqx
import jax
import jax.numpy as jnp
from jax import vmap

class DGMHJBSolver(eqx.Module):
    mlp: eqx.nn.MLP

    def __init__(self, in_size, key):
        self.mlp = eqx.nn.MLP(in_size, 1, width_size=64, depth=4, key=key, activation=jax.nn.tanh)

    def __call__(self, t, x):
        t = jnp.array([t]) if jnp.ndim(t) == 0 else t
        tx = jnp.concatenate([t, x])
        return self.mlp(tx)[0]

def loss_hjb(model, t_batch, x_batch, hamiltonian_fn, terminal_cond_fn, boundary_cond_fn, T):
    def residual(t_val, x_val):
        v_t = jax.grad(lambda _t: model(_t, x_val))(t_val)
        v_x = jax.grad(lambda _x: model(t_val, _x))(x_val)
```

```
        v_xx = jax.hessian(lambda _x: model(t_val, _x))(x_val)
        return v_t + hamiltonian_fn(x_val, v_x, v_xx)

    residuals = vmap(residual)(t_batch, x_batch)
    loss_interior = jnp.mean(residuals**2)

    v_terminal_pred = vmap(lambda x: model(T, x))(x_batch)
    v_terminal_target = vmap(terminal_cond_fn)(x_batch)
    loss_terminal = jnp.mean((v_terminal_pred - v_terminal_target)**2)

    loss_boundary = 0.0
    if boundary_cond_fn is not None:
        loss_boundary = 0.0

    return loss_interior + loss_terminal + loss_boundary
```

3.3 Branch B: Entropy Monitoring

```
import jax.numpy as jnp
from jax import vmap

def compute_entropy_dgm(model, t, x_samples, num_bins=50):
    v_values = vmap(lambda x: model(t, x))(x_samples)
    hist, bin_edges = jnp.histogram(v_values, bins=num_bins, density=True)
    bin_width = bin_edges[1] - bin_edges[0]
    probs = hist * bin_width
    log_probs = jnp.log(probs + 1e-10)
    entropy = -jnp.sum(probs * log_probs)
    return entropy
```

3.4 Branch C: IMEX Scheme

```
import jaxopt
import jax.numpy as jnp

def imex_step(x_curr, dt, drift_stiff, jump_kernel_fft, diffusion, key):
    noise = random.normal(key, x_curr.shape) * jnp.sqrt(dt)
    jump_term = compute_jump_fft(x_curr, jump_kernel_fft)
    explicit_part = x_curr + dt * jump_term + diffusion(x_curr) * noise

    def fixed_point_op(y, _):
        return explicit_part + dt * drift_stiff(y)

    solver = jaxopt.AndersonAcceleration(fixed_point_op, maxiter=10, tol=1e-5)
    x_new, _ = solver.run(x_curr, None)
    return x_new
```

3.5 Branch D: Log-Signatures

```
import signax

def compute_features(path, depth=3):
    signature = signax.signature(path, depth)
    log_sig = signax.logsignature(path, depth)
    return log_sig
```

Chapter 4

Module 3: JKO Orchestrator

4.1 Stop-Gradient for Diagnostic Modules

SIA and CUSUM are diagnostics. Enforce:

```
import jax
raw_holder = self.sia.estimate_holder_exponent(self.signal_buffer)
meta_state_h = jax.lax.stop_gradient(raw_holder)
raw_alarm, raw_kurtosis = self._check_regime_change_with_kurtosis(last_error)
regime_changed = jax.lax.stop_gradient(raw_alarm)
```

4.2 AOT Compilation Cache

Use persistent compilation cache for hot-starts:

```
import os
import jax

cache_dir = os.path.expanduser("~/.jax_cache")
os.makedirs(cache_dir, exist_ok=True)

jax.config.update('jax_compilation_cache_dir', cache_dir)
```

4.3 Warm-Up Pass

Execute a full dummy pass before opening market sockets to transfer XLA kernels to GPU and initialize CUDA context.

4.4 Deterministic Matmul Precision

Force float32 determinism:

```
import jax
jax.config.update("jax_default_matmul_precision", "highest")
```

Chapter 5

Walk-Forward Validation and Bayesian Tuning

5.1 Walk-Forward Validator

Use rolling windows without look-ahead. Vectorize windows with `jax.vmap` when models are stateless.

5.2 Optuna Meta-Optimization

Use TPE to optimize hyperparameters (signature depth, ε, τ, CUSUM parameters, Besov cone).

Chapter 6

Telemetry and Flags

Expose $\mathbb{S}_{risk}$ as a structured telemetry object, including Holder exponent, CUSUM drift, kurtosis, adaptive threshold, kernel weights, and operational flags.

Chapter 7

Strict Dependency Pinning

All environments must pin exact versions (JAX, OTT-JAX, Signax, Equinox, Diffrax, NumPy, SciPy). Open version ranges and dynamic upgrades are forbidden.

7.1 Platform-Specific Dependencies

JAX and JAXlib require platform-specific binaries due to XLA compilation targets. The `requirements.txt` uses PEP 508 environment markers to specify platform-conditional versions:

```
# macOS Intel (x86_64)
jax==0.4.38; sys_platform == 'darwin' and platform_machine == 'x86_64'
jaxlib==0.4.38; sys_platform == 'darwin' and platform_machine == 'x86_64'

# macOS ARM64 (M1/M2/M3/M4)
jax==0.4.38; sys_platform == 'darwin' and platform_machine == 'arm64'
jaxlib==0.4.38; sys_platform == 'darwin' and platform_machine == 'arm64'

# Linux (all architectures)
jax==0.4.38; sys_platform == 'linux'
jaxlib==0.4.38; sys_platform == 'linux'

# Windows
jax==0.4.38; sys_platform == 'win32'
jaxlib==0.4.38; sys_platform == 'win32'
```

Installation: The command `pip install -r requirements.txt` automatically selects the appropriate version based on the detected platform. No manual intervention required.

CI/CD Compatibility: Environment markers ensure identical deployment commands across all platforms while maintaining strict version pinning.