Why tAXIOM: Clear Experiments, Trusted Results

The Problem

Your colleague reports their method beats the baseline by 3%. You try to reproduce it. The results don't match.

Was it the learning rate? The batch size? A different random seed? Some data preprocessing they forgot to mention? A "temporary" debugging change that became permanent?

Without clear experimental protocols, we spend more time debugging reproducibility than advancing the science.

The Solution

tAXIOM makes your entire experiment—not just your model, but the complete experimental protocol—explicit, shareable, and verifiable through typed specifications and composable pipelines.

Three Core Use Cases

tAXIOM has three main use cases that make it stand out in a research environment:

1. Experiment Sharing

Pipelines act as experimental roadmaps where custom implementations can be shared between teams. Exit checks at each join ensure control variables work across platforms. ExperimentSpecs fully define experiments and make their purpose obvious—think of them as "minimal configs for scientific validation."

Example scenario: "We found that longer BPTT sequences dramatically improved curriculum learning convergence. Here's our ExperimentSpec—can you reproduce this? Our pipeline has a `require_join` for the trainer so you can test if this holds with RLlib instead of PufferLib."

→ You take their spec, swap in RLlib's PPO, and run. Any other changes (GPU count, batch size) are automatically documented in the manifest. You know exactly what's comparable and what's not.

2. Experimental Control

The `require_join` placeholder clearly defines what varies in an experiment. `provide_join` swaps components while keeping everything else untouched, letting you confidently pinpoint cause and effect.

Example scenario: "My new trainer works great on CartPole and Pendulum, but I haven't tested the full benchmark suite. Here's my pipeline with `require_join('make_env')`—just plug in your environments."

→ You plug in your env, run the experiment, and when comparing results you know *exactly* what changed—just the environment, nothing else.

3. Painless Benchmarking and A/B Testing

Iterate over different components and plug them into pipeline slots. Since you're using the exact same pipeline structure, A/B test results aren't polluted by accidental implementation differences.

Example scenario: Testing 5 loss functions × 3 augmentation strategies × 4 schedulers = 60 combinations. With tAXIOM, it's a triple nested loop where each combination runs through the identical pipeline—no hidden variations.

The Five Mechanisms

While designed for experiments, tAXIOM's syntax also declutters production RL code. Five mechanisms work together to create self-documenting code with clear data flow:

1. Stages - Pure Computation

Deterministic transformations that take values and return values. No side effects, no external dependencies—just computation.

```
.stage("normalize", lambda v: (v["obs"] - mean) / std)
.stage("compute_advantage", lambda v: v["rewards"] - v["baseline"])
```

2. I/O - External Operations

Explicitly marked operations that interact with the outside world: files, networks, databases, or mutable state.

```
.io("load_checkpoint", lambda: torch.load("model.pt"))
.io("fetch_metrics", lambda: wandb.Api().get_metrics())
```

3. Checks - Runtime Validation

Assertions at stage boundaries that verify data integrity without affecting outcomes. Can be sampled for noisy domains.

```
.through(checks={
    "valid_probs": WARN(prob_simplex(1e-6)),
    "no_nans": FAIL(no_nan()),
    "gradients_stable": WARN(grad_band(1e-6, 100), sample_every=10)
})
```

4. Hooks - Observation Points

Removable observers for logging, metrics, and debugging that never affect the pipeline's computation.

```
.through(hooks=[
    lambda v, ctx: logger.info(f"Step {ctx.step}: loss={v['loss']:.4f}"),
    lambda v, ctx: wandb.log({"score": v["score"]})
])
```

5. Guards - Execution Control

Decorators that control when and how functions execute without modifying their logic.

```
@timeout(60)
@gpu_required
@retry_on_failure(3)
def expensive_computation(data):
    return model.forward(data)
```

Together, these mechanisms create code where data flow is explicit, boundaries are clear, and experiments are reproducible by design.

A Real Example: Testing Augmentation Strategies

Your team's model plateaued at 89% accuracy. Two researchers have competing hypotheses:

Alice: "We need aggressive augmentation"
Bob: "We need better learning rate scheduling"

The Traditional Approach: Uncertainty

Alice modifies her training script with new augmentation. Bob creates his own script with a new scheduler. They also make other "minor" changes—different batch sizes, different validation splits, different random seeds.

When their results differ, you can't tell what actually helped.

The tAXIOM Approach: Controlled Experiments

```
from pydantic import BaseModel
from typing import Callable

class AugmentConfig(BaseModel):
    strength: float = 0.5
    prob: float = 0.8

class SchedulerConfig(BaseModel):
    initial_lr: float = 0.001
    decay_rate: float = 0.95

class VisionExperimentSpec(BaseModel):
    """Specification for vision training experiments."""
    # Variable components (what we're testing)
    augmentation_block: Callable[[AugmentConfig], Pipeline]
    scheduler_block: Callable[[SchedulerConfig], Pipeline]

    # Configuration for the blocks
    augment_cfg: AugmentConfig = AugmentConfig()
    scheduler_cfg: SchedulerConfig = SchedulerConfig()

    # Fixed parameters (controlled across all tests)
    dataset: str = "imagenet"
    model: str = "resnet50"
    batch_size: int = 128
    num_epochs: int = 100
    seed: int = 42

def make_vision_pipeline(spec: VisionExperimentSpec) -> Pipeline:
    """The experimental protocol - identical except for what we're testing."""
    return (Pipeline()
        .io("load", lambda: load_dataset(spec.dataset))
        .stage("init_model", lambda: create_model(spec.model))
        .join("augment", sub=spec.augmentation_block(spec.augment_cfg))
        .join("schedule_lr", sub=spec.scheduler_block(spec.scheduler_cfg))
        .stage("train_epoch", standard_training_loop)
        .io("evaluate", compute_metrics)
    )

# Alice's experiment
alice_spec = VisionExperimentSpec(
    augmentation_block=lambda cfg: Pipeline()
        .stage("augment", lambda v: aggressive_augmentation(v, strength=cfg.strength)),
    augment_cfg=AugmentConfig(strength=0.8),
    scheduler_block=lambda cfg: Pipeline()
```

```
        .stage("lr", lambda v: baseline_scheduler(v, initial_lr=cfg.initial_lr))
)

# Bob's experiment
bob_spec = VisionExperimentSpec(
    augmentation_block=lambda cfg: Pipeline()
        .stage("augment", lambda v: minimal_augmentation(v, prob=cfg.prob)),
    scheduler_block=lambda cfg: Pipeline()
        .stage("lr", lambda v: adaptive_scheduler(v, cfg))

# Run with identical protocols
alice_result = make_vision_pipeline(alice_spec).run()
bob_result = make_vision_pipeline(bob_spec).run()
```

Now you know exactly what made the difference. The `ExperimentSpec` guarantees everything else is identical.

Key Capabilities

1. Controlled Comparisons

```
# These pipelines differ in EXACTLY one place
baseline = protocol.provide_join("optimizer", sgd())
experimental = protocol.provide_join("optimizer", adam())
```

The protocol ensures everything else remains identical.

2. Systematic Ablation Studies

```
# What does each component contribute?
with_attention = make_pipeline(use_attention=True)
without_attention = make_pipeline(use_attention=False)

with_dropout = make_pipeline(use_dropout=True)
without_dropout = make_pipeline(use_dropout=False)

# Test all combinations systematically
for attention in [True, False]:
    for dropout in [True, False]:
        pipeline = make_pipeline(attention, dropout)
        results[(attention, dropout)] = pipeline.run()
```

3. Share Experimental Protocols

Sometimes you want to share just a pipeline—not a full ExperimentSpec—to test your exact setup with different components. This is where `require_join` shines:

```
# Scenario: You've developed a novel training approach and want to test
# how it performs across different environments

def our_training_protocol() -> Pipeline:
    """Our exact training setup - we want to test on different envs."""
    return (Pipeline()
        # This MUST be provided by whoever runs the pipeline
```

```
        .require_join("make_env",
                      exit_checks=[env_has_proper_spaces(),
                                   supports_reset()])

        # Our specific training approach (fixed)
        .stage("init", lambda: init_our_special_model())
        .stage("train", lambda v: our_proprietary_training(v,
                                                           lr=0.0003,
                                                           batch_size=256,
                                                           special_trick=True))
        .through(checks=[training_converged()])
        .io("evaluate", run_standard_eval)
        .through(checks=[score_in_range(0, 1)])
    )

# What others do: provide their environment implementation
their_pipeline = (our_training_protocol()
    .provide_join("make_env", lambda: Pipeline()
        .io("create", lambda: gym.make("Ant-v4"))
        .stage("wrap", lambda env: add_reward_shaping(env))
    ))

results_on_ant = their_pipeline.run()

# Test on multiple environments
for env_name in ["Ant-v4", "Humanoid-v4", "HalfCheetah-v4"]:
    pipeline = (our_training_protocol()
        .provide_join("make_env", lambda: Pipeline()
            .io("create", lambda: gym.make(env_name))))
    results[env_name] = pipeline.run()
```

The `require_join`/`provide_join` pattern is perfect for: - **Publishing research methods**: Share your training innovation, let others test on their domains - **Environment generalization studies**: Fix the trainer, vary the environment - **Benchmark creation**: Define the protocol, let implementations compete

For more complex scenarios with multiple swappable parts, the ExperimentSpec approach provides better type safety and documentation.

Systematic Experimentation in Practice

Our team needed to test 5 loss functions, 3 augmentation strategies, and 4 learning rate schedules—60 combinations total.

Traditional approach

Creating separate scripts for each combination led to inconsistencies and made it nearly impossible to identify interaction effects.

With tAXIOM

```
class LossConfig(BaseModel):
    weight_decay: float = 0.01
    label_smoothing: float = 0.1
```

```python
class GridSearchSpec(BaseModel):
    loss_block: Callable[[LossConfig], Pipeline]
    aug_block: Callable[[AugmentConfig], Pipeline]
    schedule_block: Callable[[SchedulerConfig], Pipeline]

    # Configurations
    loss_cfg: LossConfig = LossConfig()
    aug_cfg: AugmentConfig = AugmentConfig()
    schedule_cfg: SchedulerConfig = SchedulerConfig()

# Systematic grid search
results = {}
for loss_name, loss_fn in losses.items():
    for aug_name, aug_fn in augmentations.items():
        for sched_name, sched_fn in schedules.items():
            spec = GridSearchSpec(
                loss_block=lambda cfg: Pipeline()
                    .stage("loss", lambda v: loss_fn(v, weight_decay=cfg.weight_decay)),
                aug_block=lambda cfg: Pipeline()
                    .stage("aug", lambda v: aug_fn(v, strength=cfg.strength)),
                schedule_block=lambda cfg: Pipeline()
                    .stage("schedule", lambda v: sched_fn(v, lr=cfg.initial_lr)),
                loss_cfg=LossConfig(weight_decay=0.01 if loss_name == "focal" else 0.001),
                aug_cfg=AugmentConfig(strength=0.3 if loss_name == "focal" else 0.8)
            )
            pipeline = make_training_pipeline(spec)
            results[(loss_name, aug_name, sched_name)] = pipeline.run()

# Analyze interactions
interaction_matrix = compute_interactions(results)
# Discovery: focal loss works best with mild augmentation
# but cross-entropy needs aggressive augmentation
```

The key insight: certain losses paired better with certain augmentation strategies—something we would have missed without systematic testing.

Production Use Cases

Beyond research, tAXIOM provides value in production systems where code changes frequently:

1. **Clear boundaries** - Failures are localized to specific pipeline stages
2. **Component swapping** - Update algorithms without modifying infrastructure
3. **A/B testing** - Deploy changes with explicit control variables
4. **Audit trails** - Manifests document exactly what ran

Design Philosophy

tAXIOM makes explicit experiment design the natural approach:

- **Testing variations**: Swap implementations at `.join()` points
- **Ensuring correctness**: Add checks at stage boundaries
- **Debugging**: The pipeline shows the complete data flow
- **Sharing**: The ExperimentSpec and pipeline are the shareable artifacts

Incremental Adoption

You don't need to rewrite everything. Start by wrapping one existing experiment:

```
# Your existing code
def train_model(config):
    # ... 500 lines of training code ...
    return metrics

# Wrap it in tAXIOM (5 minutes)
pipeline = (Pipeline()
    .stage("train", lambda: train_model(config))
    .through(checks=[score_improved()])
)

# Now you can A/B test
config_a = config.copy(); config_a["lr"] = 0.001
config_b = config.copy(); config_b["lr"] = 0.01

pipeline_a = Pipeline().stage("train", lambda: train_model(config_a))
pipeline_b = Pipeline().stage("train", lambda: train_model(config_b))
```

The Core Value Proposition

For Researchers

- Know exactly what changed between experiments
- Share complete experimental protocols, not just model code
- Identify interaction effects between components
- Generate reproducible results with manifests

For Engineers

- Clear boundaries make debugging straightforward
- Swap components without touching the system
- A/B test with actual control variables
- Every run generates an audit trail

Getting Started

The best way to understand tAXIOM is to wrap one existing experiment:

```
from pydantic import BaseModel
from metta.sweep.axiom import Pipeline
from typing import Callable

class TrainConfig(BaseModel):
    lr: float = 0.001
    batch_size: int = 32

class MyExperimentSpec(BaseModel):
    """Your experiment configuration."""
    # Swappable component with configuration
    optimizer_block: Callable[[TrainConfig], Pipeline]

    # Configuration
```

```
    train_cfg: TrainConfig = TrainConfig()

    # Fixed parameters
    model: str = "resnet50"
    dataset: str = "cifar10"

def make_my_pipeline(spec: MyExperimentSpec) -> Pipeline:
    """Your experimental protocol."""
    return (Pipeline()
        .io("load", lambda: load_data(spec.dataset))
        .stage("init", lambda: init_model(spec.model))
        .join("optimize", sub=spec.optimizer_block(spec.train_cfg))
        .through(checks=[score_valid()])
    )

# Now you can systematically test variations
baseline_spec = MyExperimentSpec(
    optimizer_block=lambda cfg: Pipeline()
        .stage("train", lambda v: sgd_train(v, lr=cfg.lr)),
    train_cfg=TrainConfig(lr=0.001)
)

adam_spec = MyExperimentSpec(
    optimizer_block=lambda cfg: Pipeline()
        .stage("train", lambda v: adam_train(v, lr=cfg.lr)),
    train_cfg=TrainConfig(lr=0.0001)  # Adam typically uses lower LR
)

baseline = make_my_pipeline(baseline_spec)
adam = make_my_pipeline(adam_spec)
```

FAQ: Anticipated Pushback (and Honest Answers)

"Refactoring my loop into a pipeline seems like work."

It is—and it mirrors the scientific method: isolate what changes, make boundaries explicit, and encode assumptions as checks. Once you've done it once, the map is shareable.

"Do I have to use your Experiment class?"

No. It's a pattern that plays beautifully with Pydantic and clarifies what's open vs. fixed. If you want to wire pipelines by hand, go for it.

"Are we trying to do too much?"

There's a predictable critique of frameworks like this: "Isn't this over-engineering?" Here's the honest answer:

We're not pushing tAXIOM as an all-in-one coding paradigm for ML/RL. Pipelines are modular, contract-bound units of computational logic. We orchestrate them however we want—outside the pipeline. That alone removes the heaviest design burden: control-flow syntax.

The remaining features—hooks, guards, checks—aren't "extras." They are the small, sharp tools that make separation of concerns enforceable in RL/ML: - **Checks** are on-the-fly control variables. They can be trivial (same dtype) or critical (no NaNs, probability simplex), and they cannot change outcomes. - **Hooks** give uniform observability and never affect results. - **Guards** (timeouts, placement) wrap compute without touching it.

A platform that preaches a rigid separation of concerns is incomplete unless it also gives you the tools to enforce that separation. tAXIOM provides exactly those tools—no more, no less.

Learn more: tAXIOM Documentation → | Complete A/B Testing Example →