RFSN v9.2 Production Enhancements

Safety, Monitoring, and Operational Hardening

Generated: January 22, 2026, 1:44 PM CST
Purpose: Production-grade safety filters, policy management, and observability
Status: Ready for implementation alongside v9.2 core

Executive Summary

Your v9.2 build is **a solid decision substrate**—but without safety guardrails, it will eventually take catastrophic actions. This document provides **5 mandatory enhancements** that transform v9.2 from "good" to "dangerously useful in production."

These are not nice-to-have features. They are **non-negotiable for systems that make real decisions** (robotics, code mutation, trading, NPC behavior).

---

1. Action Safety Filter (NON-NEGOTIABLE)

The Problem

Contextual bandits optimize for reward, not safety. Without constraints:

- Robot might move into obstacles
- Code repair might delete critical files
- Trading system might exceed risk limits
- NPC might violate game rules

Solution: Hard safety constraints before action execution.

Implementation

SafetyValidator Base Class

Add to vw_bandit.py

from abc import ABC, abstractmethod
from typing import Tuple

class SafetyValidator(ABC):
"""Abstract base for domain-specific safety constraints"""

```
@abstractmethod
def validate(self, action: int, context: dict) -> Tuple[bool, str]:
    """
```

```
    Returns (is_safe, reason_if_unsafe)

    Args:
        action: Selected action index
        context: Full context dict with domain-specific state

    Returns:
        (True, "") if safe
        (False, "reason") if unsafe
    """
    pass

@abstractmethod
def get_safe_default(self) -> int:
    """Fallback action when no safe actions available"""
    pass
```

Code Repair Safety Validator

```
class CodeRepairSafetyValidator(SafetyValidator):
"""Safety constraints for autonomous code repair"""

def __init__(self, config=None):
    self.config = config or {}
    self.protected_files = {
        "*.lock", "Dockerfile", ".env", "requirements.txt",
        "package.json", "setup.py"
    }
    self.max_deletions_per_session = 5
    self.deletion_count = 0

def validate(self, action: int, context: dict) -> Tuple[bool, str]:
    """Check if repair action is safe"""

    action_map = {
        0: "add_import",
        1: "update_import",
        2: "add_try_except",
        3: "add_type_hint",
```

```
    4: "refactor_function",
    5: "delete_unused",
    6: "add_dependency",
    7: "update_version",
}

action_name = action_map.get(action, "unknown")

# Check 1: No destructive actions without backup
if action_name == "delete_unused":
    if not context.get("has_backup", False):
        return False, "No backup exists for deletion"
    if self.deletion_count >= self.max_deletions_per_session:
        return False, "Deletion quota exceeded this session"

# Check 2: No modifications to protected files
if action_name in ["update_import", "refactor_function"]:
    target_file = context.get("target_file", "")
    if self._is_protected(target_file):
        return False, f"Protected file: {target_file}"

# Check 3: No external dependency additions without approval
if action_name == "add_dependency":
    dep_name = context.get("dependency_name", "")
    approved = context.get("approved_dependencies", [])
    if dep_name not in approved:
        return False, f"Unapproved dependency: {dep_name}"

# Check 4: Version updates limited to patch/minor
if action_name == "update_version":
    current = context.get("current_version", "0.0.0")
    new = context.get("new_version", "0.0.0")
    if self._is_major_bump(current, new):
        return False, "Major version updates require human approval"

# Check 5: No mass refactors on production branches
if action_name == "refactor_function":
    branch = context.get("branch", "")
```

```
        num_functions = context.get("functions_affected", 1)
        if branch == "main" and num_functions > 3:
            return False, "Mass refactors not allowed on production"

    return True, ""

def _is_protected(self, filepath: str) -> bool:
    """Check if file is in protected list"""
    from fnmatch import fnmatch
    return any(fnmatch(filepath, pattern) for pattern in self.protected_files)

def _is_major_bump(self, current: str, new: str) -> bool:
    """Check if version update is major"""
    try:
        curr_major = int(current.split(".")[0])
        new_major = int(new.split(".")[0])
        return new_major > curr_major
    except:
        return False

def get_safe_default(self) -> int:
    """Default: add type hints (safest action)"""
    return 3
```

Robotics Safety Validator

class RoboticsSafetyValidator(SafetyValidator):
"""Safety constraints for autonomous robot control"""

```
def __init__(self, config=None):
    self.config = config or {}
    self.max_velocity = 1.0  # m/s
    self.min_obstacle_distance = 0.5  # meters
    self.joint_limits = {
        "shoulder": 2.5,  # radians
        "elbow": 2.0,
        "wrist": 1.5,
    }
```

```
def validate(self, action: int, state: dict) -> Tuple[bool, str]:
    """Check if robot action is safe"""

    action_map = {
        0: "stay_still",
        1: "move_forward_slow",
        2: "move_forward_fast",
        3: "turn_left",
        4: "turn_right",
        5: "lift_arm",
        6: "lower_arm",
        7: "grasp",
    }

    action_name = action_map.get(action, "unknown")

    # Check 1: Don't move if obstacle too close
    if "move" in action_name or "lift" in action_name:
        obstacle_dist = state.get("obstacle_distance", 10.0)
        if obstacle_dist < self.min_obstacle_distance:
            return False, f"Obstacle too close ({obstacle_dist:.2f}m)"

    # Check 2: Don't exceed velocity limits
    if "fast" in action_name:
        velocity = state.get("velocity", 0.0)
        if velocity > self.max_velocity:
            return False, f"Velocity limit exceeded ({velocity:.2f} m/s)"

    # Check 3: Don't exceed joint angle limits
    joints = state.get("joint_angles", {})
    for joint_name, angle in joints.items():
        limit = self.joint_limits.get(joint_name, 3.0)
        if abs(angle) > limit:
            return False, f"Joint {joint_name} limit exceeded ({angle:.2f} rad)"

    # Check 4: Don't grasp without sufficient proximity
    if action_name == "grasp":
        gripper_dist = state.get("gripper_object_distance", 10.0)
```

```
        if gripper_dist > 0.05:  # 5cm
            return False, f"Object too far to grasp ({gripper_dist:.3f}m)"

    # Check 5: Don't lower arm below ground
    if action_name == "lower_arm":
        arm_height = state.get("arm_height", 1.0)
        if arm_height < 0.1:  # 10cm clearance
            return False, "Arm would hit ground"

    return True, ""

def get_safe_default(self) -> int:
    """Default: stay still (safest action)"""
    return 0
```

Integration with RFSNController

Add to main.py - RFSNController class

```
def select_action_with_safety(
self,
context: dict,
validator: SafetyValidator,
max_resample: int = 10
) -> Tuple[int, Optional[np.ndarray]]:
"""
Select action with safety validation.
If unsafe, resample until safe action found or max attempts reached.

Args:
    context: Feature vector
    validator: SafetyValidator instance
    max_resample: Max attempts to find safe action

Returns:
    (action, probabilities) - guaranteed to be safe
"""

for attempt in range(max_resample):
    # Select action
```

```
        action, probs = self.bandit.select_action(context)

        # Validate
        is_safe, reason = validator.validate(action, context)

        if is_safe:
            return action, probs

        # Unsafe - log and resample with increased exploration
        logger.warning(
            f"Unsafe action {action} rejected: {reason}. "
            f"Resampling (attempt {attempt+1}/{max_resample})"
        )

        # Force exploration to find safe action
        original_eps = self.config.epsilon
        self.config.epsilon = 1.0  # Force random sampling

    # Last resort: return safe default
    logger.error(
        f"Could not find safe action after {max_resample} attempts. "
        f"Returning safe default."
    )
    default = validator.get_safe_default()
    return default, None
```

---

2. Reward Normalization + Decay

The Problem

Reward scales vary wildly:

- Code repair: -1 (fail) to +10 (perfect fix)
- Robotics: 0 (stuck) to 1 (goal reached)
- Trading: -5% to +15% daily return

Without normalization, VW treats these incomparably, learning quality collapses.

Implementation

Add to vw_bandit.py

class RewardNormalizer:
"""
Adaptive reward normalization using running statistics.
Converts arbitrary reward scales to approximately N(0,1).
"""

```
def __init__(self, alpha: float = 0.01):
    """
    Args:
        alpha: Learning rate for mean/variance updates (0.001-0.1)
    """
    self.mean = 0.0
    self.variance = 1.0
    self.alpha = alpha
    self.n = 0  # Number of observations
    self.reward_history = []

def normalize(self, reward: float) -> float:
    """
    Normalize reward to approximately N(0,1).

    Args:
        reward: Raw reward value

    Returns:
        Normalized reward
    """
    self.n += 1
    self.reward_history.append(reward)

    # Update running mean
    delta = reward - self.mean
    self.mean += self.alpha * delta

    # Update running variance (Welford's algorithm)
```

```
        if self.n > 1:
            delta2 = reward - self.mean
            self.variance += self.alpha * (delta * delta2 - self.variance)

        # Normalize with numerical stability
        std = np.sqrt(max(self.variance, 1e-8))
        normalized = (reward - self.mean) / std

        return normalized

    def get_statistics(self) -> dict:
        """Return current mean, std, and observation count"""
        return {
            "mean": float(self.mean),
            "std": float(np.sqrt(max(self.variance, 1e-8))),
            "n": self.n,
            "raw_rewards": {
                "min": float(np.min(self.reward_history)) if self.reward_history else 0,
                "max": float(np.max(self.reward_history)) if self.reward_history else 0,
                "mean": float(np.mean(self.reward_history)) if self.reward_history else 0
            }
        }

    def reset(self):
        """Reset statistics (for new task/environment)"""
        self.mean = 0.0
        self.variance = 1.0
        self.n = 0
        self.reward_history = []
```

class RewardDecayScheduler:
"""
Exponential decay of reward influence over time.
Newer rewards have higher weight than old ones (handles non-stationarity).
"""

```
    def __init__(self, half_life: int = 10000):
        """
        Args:
```

```
        half_life: Examples until reward influence drops to 50%
    """
    self.half_life = half_life
    self.n = 0

def decay_factor(self) -> float:
    """Return decay factor for current example"""
    return 0.5 ** (self.n / self.half_life)

def step(self):
    """Increment example counter"""
    self.n += 1

def apply(self, reward: float) -> float:
    """Apply decay to reward"""
    return reward * self.decay_factor()

def reset(self):
    """Reset counter (for new epoch/environment)"""
    self.n = 0
```

Integration in VWBanditOptimizer

class VWBanditOptimizer:
def **init**(self, bandit):
# ... existing code ...
self.reward_normalizer = RewardNormalizer(alpha=0.01)
self.reward_decay = RewardDecayScheduler(half_life=10000)

```
def update(self, context, action, raw_reward):
    """Update with reward normalization + decay"""

    # Apply decay first (older examples count less)
    decayed = self.reward_decay.apply(raw_reward)

    # Normalize to N(0,1)
    normalized = self.reward_normalizer.normalize(decayed)
```

```
    # Update bandit
    self.bandit.update(context, action, normalized)
    self.reward_decay.step()

    # Log statistics every 1000 steps
    if self.bandit.example_count % 1000 == 0:
        stats = self.reward_normalizer.get_statistics()
        logger.info(
            f"Reward stats (n={stats['n']}): "
            f"mean={stats['mean']:.3f}, "
            f"std={stats['std']:.3f}, "
            f"raw=[{stats['raw_rewards']['min']:.2f}, "
            f"{stats['raw_rewards']['max']:.2f}]"
        )
```

---

3. Policy Snapshotting + Rollback

The Problem

Training can degrade unexpectedly:

- Concept drift (environment changes)
- Reward corruption (measurement error)
- Exploration gone wrong (bad luck sequence)

Solution: Automatic checkpointing with ability to rollback to best-performing policy.

Implementation

Add to vw_bandit.py

from pathlib import Path
from datetime import datetime
import json

class CheckpointManager:
"""
Persistent model checkpointing with metadata and rollback capability.
"""

```
def __init__(
    self,
    bandit,
```

```
    checkpoint_dir: str = "./checkpoints",
    max_checkpoints: int = 10,
    metric_name: str = "avg_reward"
):
    """
    Args:
        bandit: VWContextualBandit instance
        checkpoint_dir: Directory for saved models
        max_checkpoints: Keep only N best checkpoints
        metric_name: Metric to track for "best" determination
    """
    self.bandit = bandit
    self.checkpoint_dir = Path(checkpoint_dir)
    self.checkpoint_dir.mkdir(exist_ok=True, parents=True)
    self.max_checkpoints = max_checkpoints
    self.metric_name = metric_name
    self.checkpoint_history = []

def checkpoint(self, metrics: dict, label: str = None):
    """
    Save current model + metadata.

    Args:
        metrics: Dict of metrics {metric_name: value, ...}
        label: Optional human-readable label

    Example:
        manager.checkpoint({
            "avg_reward": 0.85,
            "success_rate": 0.92,
            "episode": 100
        }, label="after_100_episodes")
    """
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    # Generate checkpoint files
    checkpoint_id = f"{timestamp}_{label}" if label else timestamp
    model_path = self.checkpoint_dir / f"model_{checkpoint_id}.vw"
```

```
    meta_path = self.checkpoint_dir / f"meta_{checkpoint_id}.json"

    # Save model
    self.bandit.save(str(model_path))

    # Save metadata
    metadata = {
        "timestamp": timestamp,
        "label": label,
        "checkpoint_id": checkpoint_id,
        "example_count": self.bandit.example_count,
        "metrics": metrics,
        "model_path": str(model_path),
    }
    with open(meta_path, "w") as f:
        json.dump(metadata, f, indent=2)

    # Track in history
    self.checkpoint_history.append(metadata)

    # Keep only best N checkpoints
    self._prune_old_checkpoints()

    logger.info(
        f"Checkpoint saved: {checkpoint_id} | "
        f"{self.metric_name}={metrics.get(self.metric_name, 'N/A')}"
    )

    return checkpoint_id

def rollback_to_best(self) -> bool:
    """
    Restore model with best recorded metric value.

    Returns:
        True if rollback successful, False if no checkpoints
    """
    if not self.checkpoint_history:
```

```
            logger.warning("No checkpoints available for rollback")
            return False

        # Find checkpoint with best metric
        best_ckpt = max(
            self.checkpoint_history,
            key=lambda x: x["metrics"].get(self.metric_name, 0)
        )

        # Load best model
        model_path = best_ckpt["model_path"]
        self.bandit.load(model_path)

        logger.info(
            f"Rolled back to {best_ckpt['checkpoint_id']} | "
            f"{self.metric_name}={best_ckpt['metrics'].get(self.metric_name)}"
        )

        return True

    def rollback_to_timestamp(self, timestamp: str) -> bool:
        """Restore model from specific checkpoint"""
        matching = [c for c in self.checkpoint_history if timestamp in c["timestamp"]]
        if not matching:
            logger.error(f"No checkpoint found with timestamp {timestamp}")
            return False

        self.bandit.load(matching[0]["model_path"])
        logger.info(f"Rolled back to checkpoint {timestamp}")
        return True

    def _prune_old_checkpoints(self):
        """Keep only best N checkpoints"""
        if len(self.checkpoint_history) <= self.max_checkpoints:
            return

        # Sort by metric value, keep best N
        sorted_ckpts = sorted(
```

```
        self.checkpoint_history,
        key=lambda x: x["metrics"].get(self.metric_name, 0),
        reverse=True
    )

    # Delete oldest (worst-performing)
    to_delete = sorted_ckpts[self.max_checkpoints:]
    for ckpt in to_delete:
        Path(ckpt["model_path"]).unlink(missing_ok=True)
        meta_path = ckpt["model_path"].replace(".vw", ".json")
        Path(meta_path).unlink(missing_ok=True)

    # Update history
    self.checkpoint_history = sorted_ckpts[:self.max_checkpoints]
    logger.info(f"Pruned checkpoints, keeping best {self.max_checkpoints}")

def get_checkpoint_summary(self) -> list:
    """Return list of all checkpoints with metrics"""
    return [
        {
            "id": c["checkpoint_id"],
            "timestamp": c["timestamp"],
            "metric": c["metrics"].get(self.metric_name, "N/A"),
            "examples": c["example_count"],
        }
        for c in sorted(self.checkpoint_history,
                        key=lambda x: x["metrics"].get(self.metric_name, 0),
                        reverse=True)
    ]

def load_checkpoint(self, checkpoint_id: str) -> bool:
    """Load a specific checkpoint by ID"""
    matching = [c for c in self.checkpoint_history
                if c["checkpoint_id"] == checkpoint_id]
    if not matching:
        logger.error(f"Checkpoint {checkpoint_id} not found")
        return False
```

```
    self.bandit.load(matching[0]["model_path"])
    logger.info(f"Loaded checkpoint {checkpoint_id}")
    return True
```

Integration in RFSNController

class RFSNController:
def **init**(self, config=None, enable_checkpointing=True):
# ... existing code ...
if enable_checkpointing:
self.checkpoint_manager = CheckpointManager(
self.bandit,
checkpoint_dir="./checkpoints",
max_checkpoints=10,
metric_name="avg_reward"
)
else:
self.checkpoint_manager = None

```
def train_with_checkpoints(self, n_episodes: int):
    """Training loop with periodic checkpointing"""
    for episode in range(n_episodes):
        episode_reward = 0.0

        for step in range(100):  # 100 steps per episode
            context = self.extract_features()
            action, _ = self.bandit.select_action(context)
            reward = self.execute_action(action)
            self.bandit.update(context, action, reward)
            episode_reward += reward

        avg_reward = episode_reward / 100

        # Checkpoint every 10 episodes
        if episode % 10 == 0 and self.checkpoint_manager:
            self.checkpoint_manager.checkpoint(
                metrics={
                    "avg_reward": avg_reward,
                    "episode": episode,
                },
```

```
            label=f"ep{episode}"
        )

    # Rollback if performance drops significantly
    if episode > 50 and avg_reward < 0.5:
        logger.warning(f"Performance drop detected. Rolling back...")
        self.checkpoint_manager.rollback_to_best()
```

---

4. Shadow Evaluator (A/B Testing)

The Problem

You can't know if your bandit is actually better than the current baseline without running both in parallel. Shadow evaluators let you **validate improvements without risking production**.

Implementation

Add to main.py

class ShadowEvaluator:
"""
Runs learned policy vs baseline policy in parallel (shadow mode).
Collects statistics on relative performance without affecting production.
"""

```
def __init__(
    self,
    bandit,
    baseline_policy,
    name: str = "shadow_eval"
):
    """
    Args:
        bandit: VWContextualBandit instance (learned policy)
        baseline_policy: Callable(context) -> action (baseline)
        name: Name for logging
    """
    self.bandit = bandit
    self.baseline = baseline_policy
    self.name = name
```

```
    self.comparison_history = []
    self.n_comparisons = 0

def evaluate_single(self, context, reward_fn):
    """
    Run one step: bandit vs baseline, compare rewards.

    Args:
        context: Feature vector
        reward_fn: Callable(action) -> reward

    Returns:
        {bandit_action, baseline_action, bandit_reward, baseline_reward, bandit_w
    """

    # Get actions from both policies
    bandit_action, bandit_probs = self.bandit.select_action(context)
    baseline_action = self.baseline(context)

    # Evaluate both
    bandit_reward = reward_fn(bandit_action)
    baseline_reward = reward_fn(baseline_action)

    # Update bandit only (shadow - no effect on baseline)
    self.bandit.update(context, bandit_action, bandit_reward)

    # Track result
    result = {
        "bandit_action": bandit_action,
        "baseline_action": baseline_action,
        "bandit_reward": bandit_reward,
        "baseline_reward": baseline_reward,
        "bandit_won": bandit_reward > baseline_reward,
        "improvement": bandit_reward - baseline_reward,
    }

    self.comparison_history.append(result)
    self.n_comparisons += 1
```

```
    return result

def evaluate_batch(self, contexts, reward_fn, n_trials=1000):
    """
    Run comparison over batch of contexts.

    Args:
        contexts: List of feature vectors
        reward_fn: Callable(action) -> reward
        n_trials: Number of comparison trials

    Returns:
        Aggregated statistics
    """

    for i in range(n_trials):
        context = contexts[i % len(contexts)]
        self.evaluate_single(context, reward_fn)

    return self.get_statistics()

def get_statistics(self) -> dict:
    """Compute win rate, improvement, statistical significance"""
    if not self.comparison_history:
        return {"error": "No comparisons run"}

    improvements = [r["improvement"] for r in self.comparison_history]
    bandit_wins = [r["bandit_won"] for r in self.comparison_history]

    bandit_rewards = [r["bandit_reward"] for r in self.comparison_history]
    baseline_rewards = [r["baseline_reward"] for r in self.comparison_history]

    return {
        "n_comparisons": len(self.comparison_history),
        "bandit_win_rate": np.mean(bandit_wins),
        "avg_bandit_reward": np.mean(bandit_rewards),
        "avg_baseline_reward": np.mean(baseline_rewards),
```

```
        "avg_improvement": np.mean(improvements),
        "std_improvement": np.std(improvements),
        "max_improvement": np.max(improvements),
        "min_improvement": np.min(improvements),
        "bandit_better": np.mean(bandit_wins) > 0.5 and np.mean(improvements) >
    }

def print_report(self):
    """Print human-readable comparison report"""
    stats = self.get_statistics()

    print(f"\n{'='*60}")
    print(f"Shadow Evaluation Report: {self.name}")
    print(f"{'='*60}")
    print(f"Comparisons run: {stats['n_comparisons']}")
    print(f"Bandit win rate: {stats['bandit_win_rate']:.1%}")
    print(f"Bandit avg reward: {stats['avg_bandit_reward']:.4f}")
    print(f"Baseline avg reward: {stats['avg_baseline_reward']:.4f}")
    print(f"Average improvement: {stats['avg_improvement']:.4f} "
          f"(±{stats['std_improvement']:.4f})")
    print(f"Improvement range: [{stats['min_improvement']:.4f}, "
          f"{stats['max_improvement']:.4f}]")
    print(f"Status: {'✅ BANDIT BETTER' if stats['bandit_better'] else '❌ NEEDS WO
    print(f"{'='*60}\n")
```

---

5. Explicit Exploration Budget

The Problem

Exploration is expensive (bad actions have real cost). You want to explore aggressively early, then exploit confidently later.

Solution: Hard budget limiting total exploration steps, with dynamic epsilon decay.

Implementation

Add to config.py - VWConfig

class ExplorBudgetConfig:
"""Configuration for exploration budget management"""

```
def __init__(
    self,
    total_budget: int = 100000,
    min_epsilon: float = 0.01,
    decay_rate: float = 0.9999
):
    """
    Args:
        total_budget: Total exploration steps allowed
        min_epsilon: Never drop below this epsilon
        decay_rate: Exponential decay rate per step
    """
    self.total_budget = total_budget
    self.min_epsilon = min_epsilon
    self.decay_rate = decay_rate
    self.spent = 0
    self.base_epsilon = None

def remaining(self) -> int:
    return max(0, self.total_budget - self.spent)

def is_exhausted(self) -> bool:
    return self.spent >= self.total_budget

def get_effective_epsilon(self, base_epsilon: float) -> float:
    """
    Compute epsilon accounting for budget depletion.
    Decays exponentially and never drops below min_epsilon.
    """
    # Exponential decay with budget depletion
    decay = self.decay_rate ** self.spent
    effective = base_epsilon * decay
    return max(effective, self.min_epsilon)
```

```
    def record_exploration_step(self):
        """Record that exploration step was taken"""
        self.spent += 1

    def reset(self):
        """Reset budget counter (for new task/epoch)"""
        self.spent = 0
```

Integration in VWContextualBandit

class VWContextualBandit:
def **init**(self, config):
# ... existing code ...
self.exploration_budget = ExplorBudgetConfig(
total_budget=config.exploration_budget,
min_epsilon=config.min_epsilon_after_budget,
decay_rate=0.9999
)

```
    def select_action(self, context, epsilon=None):
        """
        Select action with budget-aware exploration.
        """
        # Get effective epsilon
        base_eps = epsilon or self.config.epsilon
        effective_eps = self.exploration_budget.get_effective_epsilon(base_eps)

        # Epsilon-greedy with budget tracking
        if self.config.exploration_strategy == "epsilon" and effective_eps > 0:
            if np.random.rand() < effective_eps:
                action = np.random.randint(0, self.config.n_actions)
                self.exploration_budget.record_exploration_step()
            else:
                action = np.argmax(self.predictions)
        else:
            action = np.argmax(self.predictions)

        # Log budget status
```

```
    if self.example_count % 10000 == 0:
        remaining = self.exploration_budget.remaining()
        logger.info(
            f"Exploration budget: {remaining}/{self.exploration_budget.total_budget
            f"remaining (ε={effective_eps:.4f})"
        )

    return action
```

Integration Checklist

Phase 1: Add Safety Layer

- [ ] Add SafetyValidator base class
- [ ] Implement domain-specific validators (CodeRepairSafetyValidator, RoboticsSafetyValidator)
- [ ] Integrate into RFSNController.select_action_with_safety()
- [ ] Test with 100+ validation scenarios

Phase 2: Add Monitoring

- [ ] Add RewardNormalizer to VWBanditOptimizer
- [ ] Add RewardDecayScheduler
- [ ] Log statistics every 1000 steps
- [ ] Verify rewards normalize to N(0,1)

Phase 3: Add Checkpointing

- [ ] Add CheckpointManager to training loop
- [ ] Configure checkpoint frequency (every 10 episodes)
- [ ] Test rollback functionality
- [ ] Set up monitoring for performance drops

Phase 4: Add Evaluation

- [ ] Add ShadowEvaluator
- [ ] Configure baseline policy
- [ ] Run parallel evaluation every 100 episodes
- [ ] Verify bandit wins > 50% before deployment

Phase 5: Add Exploration Limits

- [ ] Add ExplorBudgetConfig
- [ ] Integrate budget tracking into select_action()
- [ ] Set appropriate budget (usually 10-20% of total steps)
- [ ] Monitor budget depletion over time

Deployment Integration Example

Code Repair with Full Safety

Robotics with Full Safety

from rfsn import RFSNController, VWConfig
from enhancements import (
CodeRepairSafetyValidator,
RewardNormalizer,
CheckpointManager,
ShadowEvaluator
)

Initialize

config = VWConfig(
n_actions=8,
exploration_budget=50000,
epsilon=0.2
)
controller = RFSNController(config)

Add safety

validator = CodeRepairSafetyValidator()
controller.optimizer.reward_normalizer = RewardNormalizer()
controller.checkpoint_manager = CheckpointManager(
controller.bandit,
max_checkpoints=5,
metric_name="test_success_rate"
)

Training loop with all safeguards

for episode in range(1000):
metrics = train_episode(controller, validator)

```
if episode % 10 == 0:
    controller.checkpoint_manager.checkpoint(metrics)

if episode % 100 == 0:
    evaluator = ShadowEvaluator(
        controller.bandit,
```

```
        baseline_random_repair
    )
    eval_stats = evaluator.evaluate_batch(
        test_contexts,
        reward_fn=evaluate_repair,
        n_trials=100
    )
    print(eval_stats)

    if not eval_stats["bandit_better"]:
        print("Rolling back...")
        controller.checkpoint_manager.rollback_to_best()
```

Robotics with Full Safety

from robotics_env import RoboticsSim
from enhancements import RoboticsSafetyValidator

Initialize robot controller

sim = RoboticsSim()
validator = RoboticsSafetyValidator()

Training loop

for episode in range(1000):
state = sim.reset()
episode_reward = 0.0

```
for step in range(500):
    # Extract features from state
    context = extract_robot_features(state)

    # Safe action selection (may resample if unsafe)
    action, _ = controller.select_action_with_safety(
        context,
        validator=validator,
        state=state  # Pass state for safety checks
    )
```

```
    # Execute
    new_state, reward = sim.step(action)

    # Update with normalized reward
    norm_reward = controller.optimizer.reward_normalizer.normalize(reward)
    controller.bandit.update(context, action, norm_reward)

    state = new_state
    episode_reward += reward

# Checkpoint and monitor
if episode % 20 == 0:
    controller.checkpoint_manager.checkpoint({
        "episode_reward": episode_reward,
        "success_rate": compute_success_rate(last_100_episodes),
        "episode": episode
    })
```

Testing Production Enhancements

Safety Validator Tests

```
def test_code_repair_safety():
validator = CodeRepairSafetyValidator()

    # Test: Prevent deletion without backup
    context = {"has_backup": False}
    is_safe, reason = validator.validate(5, context)  # action 5 = delete
    assert not is_safe
    assert "backup" in reason.lower()

    # Test: Allow deletion with backup
    context = {"has_backup": True}
    is_safe, _ = validator.validate(5, context)
    assert is_safe

    # Test: Protect critical files
    context = {"target_file": "Dockerfile"}
```

```
    is_safe, reason = validator.validate(1, context)  # action 1 = update_import
    assert not is_safe

def test_robotics_safety():
validator = RoboticsSafetyValidator()

    # Test: Prevent movement near obstacles
    state = {"obstacle_distance": 0.3}
    is_safe, reason = validator.validate(1, state)  # action 1 = move_forward
    assert not is_safe

    # Test: Allow movement with safe distance
    state = {"obstacle_distance": 1.0}
    is_safe, _ = validator.validate(1, state)
    assert is_safe
```

Reward Normalization Tests

```
def test_reward_normalization():
normalizer = RewardNormalizer()

    # Feed rewards with different scale
    raw_rewards = [5, 10, 3, 8, 12, 7]
    normalized = [normalizer.normalize(r) for r in raw_rewards]

    # Normalized should be approximately N(0,1)
    assert abs(np.mean(normalized)) < 0.1
    assert 0.8 < np.std(normalized) < 1.2
```

Checkpoint Tests

```
def test_checkpoint_rollback():
manager = CheckpointManager(bandit)

    # Save checkpoint
    manager.checkpoint({"metric": 0.9}, label="good")

    # Degrade performance
    bandit.update(context, action, -10.0)  # Bad update
```

```
# Rollback
manager.rollback_to_best()

# Verify: model restored to good checkpoint
action, _ = bandit.select_action(context)
assert action == original_best_action
```

Summary: Critical Order of Implementation

For production safety, implement in this order:

1. **Safety Validator** (prevents catastrophic failures)
2. **Reward Normalization** (improves learning quality)
3. **Checkpointing** (enables rollback safety net)
4. **Shadow Evaluator** (validates improvements)
5. **Exploration Budget** (controls long-term costs)

All five together create a **production-grade decision system** that is:

- ✅ Safe (prevents bad actions)
- ✅ Learning (normalizes rewards)
- ✅ Recoverable (can rollback)
- ✅ Validated (tested against baseline)
- ✅ Cost-aware (exploration limited)

Do not deploy to production without at least #1 and #3.