# Recommended Next Features - Implementation Guide

## 🎯 Top 5 Features to Add Next

Based on typical astrophotography workflows and missing critical features:

---

## #1: Auto-Focus Routine ⭐⭐⭐

**Priority:** CRITICAL
**Effort:** 3-5 days
**Impact:** Massive quality improvement

### Why You Need This:

- Focus drifts with temperature (even 1°C affects focus)
- Manual focusing wastes time and isn't repeatable
- Required for unattended imaging sessions
- Filter changes require refocus

### What It Does:

1. Takes series of exposures at different focus positions
2. Measures star sharpness (HFR - Half-Flux Radius)
3. Finds optimal focus position (minimum HFR)
4. Returns focuser to best position

### Implementation Outline:

python

```python
# autofocus.py - New file
import numpy as np
from scipy import ndimage

class AutoFocus:
    def __init__(self, camera, focuser):
        self.camera = camera
        self.focuser = focuser

    def calculate_hfr(self, image):
        """
        Calculate Half-Flux Radius of stars
        Lower HFR = sharper stars = better focus
        """
        # 1. Background subtraction
        background = np.median(image)
        image_sub = image - background

        # 2. Find stars (simple threshold)
        threshold = np.mean(image_sub) + 3 * np.std(image_sub)
        stars = image_sub > threshold

        # 3. Label connected components
        labeled, num_stars = ndimage.label(stars)

        if num_stars == 0:
            return None

        # 4. Calculate HFR for each star
        hfrs = []
        for i in range(1, num_stars + 1):
            star_mask = labeled == i

            # Skip if too small/large
            star_pixels = np.sum(star_mask)
            if star_pixels < 10 or star_pixels > 1000:
                continue

            # Find centroid
            y, x = np.where(star_mask)
            flux = image_sub[star_mask]

            cx = np.sum(x * flux) / np.sum(flux)
            cy = np.sum(y * flux) / np.sum(flux)

            # Calculate HFR
            distances = np.sqrt((x - cx)**2 + (y - cy)**2)
```

```python
        sorted_flux = np.sort(flux)[::-1]
        half_flux = np.sum(sorted_flux) / 2

        cumsum = 0
        for j, f in enumerate(sorted_flux):
            cumsum += f
            if cumsum >= half_flux:
                hfr = distances[j]
                hfrs.append(hfr)
                break

    # Return median HFR
    return np.median(hfrs) if hfrs else None

def run_vcurve(self, exposure_time=2.0,
               initial_position=None,
               step_size=500,
               num_steps=9):
    """
    V-curve autofocus: sample positions around current focus
    """
    # Get starting position
    if initial_position is None:
        initial_position = self.focuser.get_position()

    # Calculate sample positions (centered on initial)
    half_steps = num_steps // 2
    positions = [
        initial_position + (i - half_steps) * step_size
        for i in range(num_steps)
    ]

    # Ensure within bounds
    positions = [
        max(0, min(p, self.focuser.max_position))
        for p in positions
    ]

    results = []

    print(f"Starting V-curve autofocus...")
    print(f"  Initial: {initial_position}")
    print(f"  Range: {positions[0]} to {positions[-1]}")
    print(f"  Step: {step_size}")

    for i, pos in enumerate(positions):
        print(f"\n[{i+1}/{num_steps}] Testing position {pos}...")

        # Move focuser
        self.focuser.move_to(pos)
```

```
            self.focuser.move_to(pos)

            # Take exposure
            self.camera.start_exposure(exposure_time, light=True)

            # Wait for completion
            while self.camera.camera_state != 3:  # ImageReady
                time.sleep(0.1)

            # Get image
            image = self.camera.get_image_array()

            # Calculate HFR
            hfr = self.calculate_hfr(image)

            if hfr is not None:
                results.append((pos, hfr))
                print(f"  HFR: {hfr:.2f} pixels")
            else:
                print(f"  HFR: No stars detected")

        if len(results) < 3:
            print("✗ Insufficient data points for autofocus")
            return None

        # Find best position (minimum HFR)
        results.sort(key=lambda x: x[1])
        best_position, best_hfr = results[0]

        # Optional: Fit parabola for sub-step accuracy
        # (implement if needed for higher precision)

        # Move to best position
        print(f"\n✓ Best focus: {best_position} (HFR: {best_hfr:.2f})")
        self.focuser.move_to(best_position)

        return best_position

    def auto_focus(self, exposure_time=2.0, coarse=True, fine=True):
        """
        Two-stage autofocus: coarse then fine
        """
        current_pos = self.focuser.get_position()

        # Coarse focus (large steps)
        if coarse:
            print("\n=== Coarse Focus ===")
            coarse_pos = self.run_vcurve(
                exposure_time=exposure_time,
                initial_position=current_pos,
```

```
            step_size=1000,
            num_steps=7
        )
        current_pos = coarse_pos or current_pos

    # Fine focus (small steps)
    if fine:
        print("\n=== Fine Focus ===")
        fine_pos = self.run_vcurve(
            exposure_time=exposure_time,
            initial_position=current_pos,
            step_size=100,
            num_steps=9
        )
        current_pos = fine_pos or current_pos

    return current_pos
```

### Integration with main.py:

```python
# Add route for autofocus
@app.route('/api/v1/focuser/0/autofocus', methods=['PUT'])
def focuser_autofocus():
    """Run autofocus routine"""
    if not focuser or not camera_zwo:
        return helpers.alpaca_error(1024, "Focuser or camera not available")

    exposure = helpers.get_form_value('ExposureTime', 2.0, float)

    from autofocus import AutoFocus
    af = AutoFocus(camera_zwo, focuser)

    # Run in background thread
    def run():
        af.auto_focus(exposure_time=exposure)

    import threading
    thread = threading.Thread(target=run, daemon=True)
    thread.start()

    return helpers.alpaca_response()
```

### Testing:

```bash
# Test autofocus
curl -X PUT "http://localhost:5555/api/v1/focuser/0/autofocus" \
  -d "ExposureTime=2.0"
```

---

## #2: Improved Slewing Detection ⭐⭐⭐

**Priority:** HIGH
**Effort:** 1 day
**Impact:** Reliable slew completion

### Why You Need This:

- OnStepX doesn't provide reliable IsSlewing status
- Clients may proceed before slew completes
- Causes plate solve failures and tracking issues

### Implementation:

python

```python
# In telescope.py

def is_slewing(self):
    """
    Enhanced slewing detection using position stability
    """
    if not self.is_connected:
        return False

    # If we never started a slew, not slewing
    if not hasattr(self, '_slewing_target'):
        return False

    # Get current position
    current_ra = self.get_right_ascension()
    current_dec = self.get_declination()

    # Check if close to target (within 1 arcminute)
    ra_diff = abs(current_ra - self._slewing_target[0]) * 15 * 60  # to arcmin
    dec_diff = abs(current_dec - self._slewing_target[1]) * 60     # to arcmin

    threshold = 1.0  # 1 arcminute

    if ra_diff < threshold and dec_diff < threshold:
        # Close to target, check stability
        if not hasattr(self, '_stable_since'):
            self._stable_since = time.time()
            self._last_position = (current_ra, current_dec)
            return True

        # Check if position has been stable
        stable_time = time.time() - self._stable_since

        # Has position moved?
        last_ra_diff = abs(current_ra - self._last_position[0]) * 15 * 60
        last_dec_diff = abs(current_dec - self._last_position[1]) * 60

        if last_ra_diff > 0.1 or last_dec_diff > 0.1:
            # Still moving, reset stability timer
            self._stable_since = time.time()
            self._last_position = (current_ra, current_dec)
            return True

        # Position stable for 2 seconds = slew complete
        if stable_time > 2.0:
            del self._slewing_target
            del self._stable_since
```

```python
            return False

        return True

    # Not close to target yet
    return True

def slew_to_coords(self, ra_hours, dec_degrees):
    """Slew to RA/Dec with target tracking"""
    self._slewing_target = (ra_hours, dec_degrees)

    ra_str = helpers.format_ra_hours(ra_hours)
    dec_str = helpers.format_dec_degrees(dec_degrees).replace(':', '*')

    self.send_command(f':Sr{ra_str}#')
    self.send_command(f':Sd{dec_str}#')

    response = self.send_command(':MS#')
    return response == '0'
```

---

## #3: Dithering Support ⭐⭐

**Priority:** MEDIUM-HIGH
**Effort:** 2 days
**Impact:** Better image quality

### Why You Need This:

- Eliminates hot pixels and pattern noise
- Standard practice in astrophotography
- Simple but very effective

### Implementation:

python

```python
# dithering.py - New file

import random
import math

class Dithering:
    def __init__(self, telescope):
        self.telescope = telescope
        self.original_ra = None
        self.original_dec = None

    def dither(self, radius_pixels=5, pixel_scale=1.0):
        """
        Dither telescope by random offset

        Args:
            radius_pixels: Max dither radius in pixels
            pixel_scale: Arcseconds per pixel

        Returns:
            (offset_ra, offset_dec) in arcseconds
        """
        # Save original position on first dither
        if self.original_ra is None:
            self.original_ra = self.telescope.get_right_ascension()
            self.original_dec = self.telescope.get_declination()

        # Random angle and radius
        angle = random.uniform(0, 2 * math.pi)
        radius = random.uniform(0, radius_pixels) * pixel_scale

        # Convert to RA/Dec offsets (arcseconds)
        offset_ra = radius * math.cos(angle)
        offset_dec = radius * math.sin(angle)

        # Convert arcseconds to degrees
        offset_ra_deg = offset_ra / 3600.0
        offset_dec_deg = offset_dec / 3600.0

        # Convert RA offset to hours (accounting for declination)
        dec_rad = math.radians(self.original_dec)
        offset_ra_hours = offset_ra_deg / (15.0 * math.cos(dec_rad))
        offset_dec_hours = offset_dec_deg / 15.0

        # Calculate new position
        new_ra = self.original_ra + offset_ra_hours
        new_dec = self.original_dec + offset_dec_hours
```

```python
        # Slew to dithered position
        print(f"Dithering: RA={offset_ra:.1f}\" Dec={offset_dec:.1f}\"")
        self.telescope.slew_to_coords(new_ra, new_dec)

        return (offset_ra, offset_dec)

    def return_to_center(self):
        """Return to original undithered position"""
        if self.original_ra is not None:
            print("Returning to original position")
            self.telescope.slew_to_coords(
                self.original_ra,
                self.original_dec
            )
            self.original_ra = None
            self.original_dec = None
```

### Usage in N.I.N.A.:

Configure dithering in sequence:

- After every N exposures
- Radius: 3-10 pixels
- Settle time: 5-10 seconds

---

## #4: Simultaneous Camera Exposures ⭐⭐

**Priority:** MEDIUM
**Effort:** 2-3 days
**Impact:** Enable guiding while imaging

### Why You Need This:

- Guide camera needs to expose during imaging
- Currently cameras take turns
- Required for autoguiding

### Implementation:

```python
# In camera_zwo.py and camera_touptek.py

# Make exposure async with threading
def start_exposure(self, duration, light=True):
    """Start asynchronous exposure"""
    if self.camera_state != 0:  # Idle
        raise Exception("Camera busy")

    self.camera_state = 1  # Exposing
    self.exposure_start = time.time()
    self.exposure_duration = duration

    def exposure_thread():
        try:
            # Start hardware exposure
            self._start_hardware_exposure(duration, light)

            # Wait for completion
            while not self._is_exposure_complete():
                time.sleep(0.1)

            # Download image
            self._download_image()

            self.camera_state = 3  # ImageReady
        except Exception as e:
            print(f"Exposure error: {e}")
            self.camera_state = 0  # Idle

    import threading
    self.exposure_thread = threading.Thread(
        target=exposure_thread,
        daemon=True
    )
    self.exposure_thread.start()
```

Now both cameras can expose simultaneously!

---

## #5: Meridian Flip Handling ⭐⭐

**Priority:** MEDIUM
**Effort:** 2-3 days
**Impact:** All-night imaging

### Why You Need This:

- Targets cross meridian during long sessions
- Must flip to prevent cable wrap
- Requires pause, flip, resync, resume

### Implementation:

```python
# meridian_flip.py - New file

class MeridianFlipHandler:
    def __init__(self, telescope):
        self.telescope = telescope
        self.flip_threshold = 0.5  # Hours past meridian

    def should_flip(self):
        """Check if meridian flip needed"""
        ra = self.telescope.get_right_ascension()
        lst = self.telescope.get_sidereal_time()

        hour_angle = lst - ra
        if hour_angle < 0:
            hour_angle += 24

        # Should flip if > 12 + threshold hours
        return hour_angle > (12 + self.flip_threshold)

    def perform_flip(self, target_ra, target_dec):
        """Execute meridian flip"""
        print("Performing meridian flip...")

        # 1. Stop any ongoing operations
        self.telescope.stop_slew()

        # 2. Slew to target on opposite pier side
        # OnStepX handles this automatically
        self.telescope.slew_to_coords(target_ra, target_dec)

        # 3. Wait for slew completion
        while self.telescope.is_slewing():
            time.sleep(1)

        print("✓ Meridian flip complete")

        # 4. Caller should plate solve and recalibrate guiding
        return True
```

### Integration with N.I.N.A.:

N.I.N.A. handles meridian flips automatically when using Alpaca telescope. Your telescope just needs reliable IsSlewing and slew commands - which you have!

---

## 📋 Implementation Checklist

### Phase 1: Critical (1 week)

- [ ] Improved slewing detection (1 day)
- [ ] Simultaneous exposures (2-3 days)
- [ ] Auto-focus routine (3-5 days)

### Phase 2: Workflow (1 week)

- [ ] Dithering support (2 days)
- [ ] Meridian flip handling (2-3 days)
- [ ] Error recovery (3 days)

### Phase 3: Integration (1 week)

- [ ] Plate solving integration (3-4 days)
- [ ] Configuration persistence (1-2 days)
- [ ] Enhanced logging (2 days)

---

## 🎯 Quick Wins (< 1 Day Each)

1. **Configuration Persistence** - Save filter names/offsets
2. **Enhanced Logging** - Structured JSON logs
3. **Performance Metrics** - Track exposure timing
4. **Backup System** - Auto-backup config files

---

## ✅ What You Already Have

**Good news - these are DONE:**

- ✅ **Pulse Guiding** - PHD2 ready!
- ✅ **UDP Discovery** - Auto-discovery works
- ✅ **Filter Wheel** - With focus offsets
- ✅ **Focuser** - With temperature monitoring
- ✅ **Network/USB** - Flexible connection

---

## 🚀 Recommended Action Plan

**Week 1:** Auto-Focus

- Most impactful feature
- Uses existing hardware
- Immediate quality improvement

**Week 2:** Slewing + Simultaneous Exposures

- Fixes OnStepX limitation
- Enables proper guiding

**Week 3:** Dithering + Meridian Flip

- Standard workflow features
- N.I.N.A. integration

**Result:** Fully automated, professional imaging setup! 🌟

---

**Start with auto-focus - you'll see the difference immediately in your images!**