Progression System: Phased Implementation Plan

Overview

This document outlines a careful, incremental implementation of the progression system architecture. Each phase is designed to minimize risk, maintain system stability, and provide testable milestones before moving forward.

Phase 0: Foundation & Risk Assessment

Duration: 3-5 days **Goal:** Set up infrastructure without touching game logic

Tasks

1. **Create New File Structure**

```
idle_01/progression/
├── models/
│   ├── StoryStateModel.swift
│   ├── MilestoneStateModel.swift
│   ├── JournalEntryModel.swift
│   └── PlaystyleProfileModel.swift
├── managers/
│   ├── ProgressionManager.swift
│   └── StoryEngine.swift
├── story/
│   └── StoryDefinition.json
└── tests/
    └── ProgressionTests.swift
```

2. **SwiftData Schema Setup**
   - Add progression models to SwiftData container
   - Create migration strategy for existing saves
   - Test model persistence in isolation
3. **Stub Implementation**
   - Create empty managers with logging only
   - No side effects, just observation
   - Verify compilation and architecture fit

Potential Issues & Mitigations

Issue	Risk	Mitigation

Issue	Risk	Mitigation
SwiftData migration breaks existing saves	HIGH	Create separate model container for progression data; maintain independent versioning
File structure conflicts with Xcode	MEDIUM	Add files incrementally; test build after each addition
Memory overhead from new models	LOW	Profile memory before/after; use lazy loading

Success Criteria

- ✅ All new files compile without errors
- ✅ Existing game functionality unchanged
- ✅ Models persist and load correctly in isolation
- ✅ No performance regression (profile with Instruments)

Phase 1: Observation Layer (Read-Only)

Duration: 5-7 days **Goal:** Hook into existing systems WITHOUT modifying behavior

Tasks

1. **Add Progression Hooks**
   - Insert `ProgressionManager.shared.onCommand()` in TerminalCommandExecutor.swift
   - Insert `ProgressionManager.shared.onThoughtResolved()` in thought handlers
   - Insert `ProgressionManager.shared.onTick()` in SimulationEngine.swift
   - **CRITICAL:** All hooks are NO-OPs that only log/journal

2. **Journal Recording**
   - Log every command to `JournalEntryModel`
   - Log every thought lifecycle event
   - Log stat changes and thresholds crossed
   - **No game state changes yet**

3. **Playstyle Profiling**
   - Track command frequency
   - Calculate thought completion rate
   - Measure session patterns
   - Store in `PlaystyleProfileModel`

4. **Milestone Detection (Passive)**
   - Check milestone requirements silently
   - Log "would trigger" events
   - Don't unlock anything yet

Potential Issues & Mitigations

Issue	Risk	Mitigation
Hook call order affects game logic	HIGH	Place hooks AFTER all game logic completes; use `defer` blocks for safety
Journal grows unbounded	MEDIUM	Implement rolling window (keep last 1000 entries); add pruning logic
Performance impact from frequent logging	MEDIUM	Use background queue for journal writes; batch entries
Thread safety with shared ProgressionManager	HIGH	Use actor isolation for ProgressionManager; Swift 6 concurrency
Hook crashes break game	CRITICAL	Wrap all progression calls in `do-catch`; log errors but never throw

Implementation Safety Pattern

```
// SAFE HOOK PATTERN
func executeCommand(_ input: String) async -> CommandOutput {
    // 1. Execute game logic FIRST
    let output = await actualCommandLogic(input)

    // 2. Call progression hook AFTER, in protected context
    Task.detached { [weak self] in
        do {
            try await ProgressionManager.shared.onCommand(
                input: input,
                city: self?.currentCity
            )
        } catch {
            // Log but never crash
            print("Progression hook failed: \(error)")
        }
    }

    // 3. Return game result (unaffected by progression)
    return output
}
```

Success Criteria

- ✅ Game plays identically to before
- ✅ Journal captures all expected events
- ✅ Playstyle profile updates correctly
- ✅ Milestone detection logs show accurate triggers
- ✅ No crashes or performance degradation

- ✅ Thread safety verified with Thread Sanitizer

Phase 2: Story Engine (Dialogue Only)

Duration: 5-7 days **Goal:** Trigger story beats without affecting mechanics

Tasks

1. **Story Definition Authoring**
   - Create `StoryDefinition.json` with MVP content
   - Define 2-3 simple chapters
   - Write 5-10 story beats with dialogue
   - No branching yet (linear path)
2. **Story Engine Implementation**
   - Parse JSON into runtime structures
   - Validate schema on load
   - Implement beat triggering logic
   - Queue and display dialogue
3. **Narrative Integration**
   - Story beats appear as terminal output
   - Distinguish story dialogue from ambient narrative
   - Format: `[CITY]: Story dialogue here...`
   - Ambient narrative continues as fallback
4. **Beat Trigger Types**
   - `on_chapter_start` - fires when chapter begins
   - `on_milestone` - fires when milestone achieved
   - `on_stat_threshold` - fires when stat crosses value
   - Test each trigger type independently

Potential Issues & Mitigations

Issue	Risk	Mitigation
JSON parsing errors crash game	HIGH	Validate at build time; fallback to empty story on parse failure
Beat triggers fire multiple times	MEDIUM	Track triggered beats in `StoryStateModel`; idempotent checks
Dialogue spam from rapid triggers	MEDIUM	Rate-limit beats (max 1 per 5 seconds); queue overflow protection
Story state desync with game state	MEDIUM	Store story state per-city; validate consistency on load

Issue	Risk	Mitigation
Beat conditions race with game state	MEDIUM	Snapshot game state before evaluating conditions
Malformed JSON breaks story system	HIGH	Schema validation; comprehensive error messages; skip malformed beats

JSON Validation Strategy

```
final class StoryLoader {
    func loadStory() -> Result<StoryDefinition, StoryError> {
        do {
            let data = try loadJSON("StoryDefinition.json")
            let story = try JSONDecoder().decode(StoryDefinition.self,
from: data)

            // Validate structure
            try validateStory(story)

            return .success(story)
        } catch {
            logError(error)
            // Return minimal valid story as fallback
            return .success(StoryDefinition.minimal())
        }
    }

    func validateStory(_ story: StoryDefinition) throws {
        // Check for duplicate IDs
        let beatIDs = story.allBeats.map(\.id)
        let duplicates = beatIDs.duplicates
        guard duplicates.isEmpty else {
            throw StoryError.duplicateBeatIDs(duplicates)
        }

        // Verify all beat references exist
        for beat in story.allBeats {
            if let next = beat.nextBeat {
                guard story.beatExists(next) else {
                    throw StoryError.invalidBeatReference(next)
                }
            }
        }

        // More validations...
    }
}
```

Success Criteria

- ✅ Story beats trigger at correct moments
- ✅ Dialogue displays in terminal correctly
- ✅ No duplicate or missed beats
- ✅ JSON validation catches errors early
- ✅ Fallback works when story fails to load
- ✅ Story state persists across sessions

Phase 3: Milestone System (Soft Unlocks)

Duration: 5-7 days **Goal:** Activate milestone unlocks WITHOUT hard-gating content

Tasks

1. **Milestone Implementation**
    - Create 5-10 core milestones
    - Implement requirement checking
    - Trigger narrative responses
    - Apply stat modifiers
2. **Soft Unlock System**
    - Announce unlocks via dialogue
    - Visual indicators in UI (optional)
    - No blocking of commands/features
    - "You've discovered X" messaging
3. **Milestone Types**
    - `first_contact` - any command
    - `first_thought` - complete a thought
    - `stat_threshold` - coherence > 0.5
    - `command_mastery` - use command N times
    - `discovery` - find hidden insight
4. **Unlock Feedback**
    - Clear announcement when unlocked
    - Journal entry recording
    - Optional UI notification
    - Retroactive unlock for existing saves

Potential Issues & Mitigations

Issue	Risk	Mitigation
Milestone fires prematurely	MEDIUM	Test all requirement conditions thoroughly; add cooldown periods

Issue	Risk	Mitigation
Retroactive unlock floods new saves	LOW	Batch announcements; "You've discovered N new things" summary
Stat modifiers unbalance game	MEDIUM	Start with tiny values (0.01-0.05); tune based on playtesting
Milestone persistence fails	MEDIUM	Validate milestone IDs on save/load; handle missing milestones gracefully
Combinatorial requirements break	MEDIUM	Test all requirement combinations; validate requirement trees
Players miss unlock announcements	LOW	Store announcements; add "recent unlocks" command

Requirement Testing Framework

```
// Test milestone requirements in isolation
class MilestoneTests: XCTestCase {
    func testFirstContactMilestone() {
        let manager = ProgressionManager()
        let city = MockCity()

        // Before command

XCTAssertFalse(manager.isMilestoneAchieved("milestone_first_contact", for:
city))

        // Execute command
        manager.onCommand(input: "help", parsed: .help, city: city)

        // After command

XCTAssertTrue(manager.isMilestoneAchieved("milestone_first_contact", for:
city))

        // Second call doesn't retrigger
        manager.onCommand(input: "status", parsed: .status, city: city)

XCTAssertEqual(manager.milestoneAchievementCount("milestone_first_contact"
), 1)
    }

    func testStatThresholdMilestone() {
        let manager = ProgressionManager()
        let city = MockCity()

        city.stats.coherence = 0.4
        manager.evaluateMilestones(for: city)
        XCTAssertFalse(manager.isMilestoneAchieved("milestone_coherent",
```

```
for: city))

        city.stats.coherence = 0.6
        manager.evaluateMilestones(for: city)
        XCTAssertTrue(manager.isMilestoneAchieved("milestone_coherent",
for: city))
    }
}
```

Success Criteria

- ✅ All milestones trigger correctly
- ✅ Unlocks announced clearly
- ✅ No false positives/negatives
- ✅ Stat modifiers feel appropriate
- ✅ System works with existing saves
- ✅ Performance acceptable with 50+ milestones

Phase 4: Branching Narratives

Duration: 7-10 days **Goal:** Implement story branches based on playstyle

Tasks

1. **Branch Condition System**
    - Implement all condition types from architecture
    - `highTrust` / `lowTrust`
    - `highAutonomy` / `lowAutonomy`
    - `balancedStats` / `focusedStats`
    - `sessionPattern` analysis
2. **Story Variants**
    - Write 2-3 branching points
    - Different dialogue per branch
    - Track which branch was taken
    - Merge points for rejoining main story
3. **Branch Selection Logic**
    - Evaluate conditions at branch points
    - Handle multiple valid branches (priority system)
    - Prevent flip-flopping between branches
    - Store branch history
4. **Testing Different Playstyles**
    - Create test profiles for each archetype

- Verify correct branches trigger
- Test edge cases (equal stats, etc.)

Potential Issues & Mitigations

Issue	Risk	Mitigation
Branch explosion creates unmaintainable content	HIGH	Limit branches to 2-3 per junction; merge frequently
Players see "wrong" branch for their playstyle	MEDIUM	Clear condition thresholds; add hysteresis to prevent flip-flopping
Branch conditions conflict	MEDIUM	Define priority order; test all combinations
Story becomes incoherent across branches	MEDIUM	Write clear branch contexts; use variables for consistency
Save compatibility breaks with new branches	LOW	Version story state; migrate gracefully
Playtest reveals poorly balanced conditions	HIGH	Make thresholds data-driven (easy to tune); log branch decisions

Branch Design Guidelines

```
GOOD BRANCH DESIGN:
- 2-3 options max per junction
- Clear conditions (trust > 0.7 vs < 0.3)
- Merge within 3-5 beats
- Each branch feels meaningfully different
- Shared context between branches

BAD BRANCH DESIGN:
- 5+ branches (too complex)
- Overlapping conditions (ambiguous)
- Branches never merge (exponential content)
- Branches differ only in one line
- Requires remembering distant story beats
```

Branch Testing Matrix

Playstyle	Key Stats	Expected Branch	Test Result
Collaborative	trust: 0.8, autonomy: 0.6	"Partnership"	✅
Exploitative	trust: 0.2, coherence: 0.7	"Suspicion"	✅
Patient	sessionPattern: patient	"Contemplative"	✅
Balanced	All stats ~0.5	"Harmonious"	✅

Success Criteria

- ✅ Branches trigger based on playstyle
- ✅ Each branch feels distinct
- ✅ No orphaned story states
- ✅ Branch decisions logged clearly
- ✅ Tunable thresholds via JSON
- ✅ Playtesters confirm branch accuracy

Phase 5: Advanced Features

Duration: 7-10 days **Goal:** Add polish and complex interactions

Tasks

1. **Player Choice System**
    - Explicit choice prompts in terminal
    - `> [1] Answer honestly [2] Deflect [3] Say nothing`
    - Store choices in journal
    - Choices affect branches and stats
2. **Complex Requirements**
    - `combinationOf` - ALL conditions
    - `anyOf` - OR conditions
    - Nested requirements
    - Time-windowed requirements
3. **Discovery System**
    - Hidden insights via exploration
    - "Analyze" command reveals secrets
    - Discovery log/codex
    - Unlocks gated behind discoveries
4. **Memory System**
    - City references past events
    - "You asked me this before..."
    - Callback to significant moments
    - Persistent personality shifts
5. **Playstyle Adaptation**
    - City comments on your behavior
    - "You check in rarely, but thoroughly"
    - Adaptive dialogue tone
    - Rewards for consistent playstyle

Potential Issues & Mitigations

Issue	Risk	Mitigation
Choice UI breaks terminal flow	MEDIUM	Keep choices inline; clear formatting; timeout for idle choices
Complex requirements impossible to satisfy	MEDIUM	Validate requirement trees; provide debug command to check progress
Memory references confuse new players	LOW	Gate callbacks behind sufficient context; "The city seems pensive"
Discovery system feels grindy	MEDIUM	Balance hidden vs obvious; hints via ambient dialogue
Playstyle adaptation feels judgmental	LOW	Neutral tone; celebrate all playstyles; no "wrong" way to play
System complexity overwhelms testing	HIGH	Build comprehensive test suite; automated playstyle simulations

Choice Input Handling

```
// Choice prompt structure
struct ChoicePrompt {
    let context: String
    let options: [ChoiceOption]
    let timeout: TimeInterval? // Auto-select after timeout
    let defaultChoice: Int? // If timeout occurs
}

struct ChoiceOption {
    let id: String
    let text: String
    let requirements: [Requirement]? // Can be gated
    let consequences: ChoiceConsequences
}

struct ChoiceConsequences {
    let statChanges: [String: Double]
    let flagsSet: [String]
    let nextBeat: String
}

// Safe handling
func handleChoice(_ prompt: ChoicePrompt) async -> ChoiceOption {
    // Show options
    displayChoices(prompt.options)

    // Wait for input with timeout
    if let choice = await waitForInput(timeout: prompt.timeout) {
```

```
            return choice
        } else {
            // Timeout: use default or first available
            return prompt.options[prompt.defaultChoice ?? 0]
        }
    }
```

Success Criteria

- ✅ Choices work smoothly in terminal
- ✅ Complex requirements function correctly
- ✅ Discovery system feels rewarding
- ✅ Memory callbacks enhance immersion
- ✅ Playstyle adaptation is insightful
- ✅ System is thoroughly tested

Phase 6: Hard Unlocks (Optional)

Duration: 3-5 days **Goal:** Gate advanced content behind progression

Tasks

1. **Hard Gate Implementation**
    - Commands return "Not yet unlocked" if gated
    - Clear unlock conditions shown
    - Graceful degradation for missing content
2. **Progressive Disclosure**
    - Basic commands available immediately
    - Advanced commands unlock through play
    - `help` shows locked commands grayed out
3. **Unlock Pacing**
    - Core loop available from start
    - First unlock within 5-10 minutes
    - Major unlocks every 30-60 minutes
    - Endgame content after several hours

Potential Issues & Mitigations

Issue	Risk	Mitigation
Hard gates frustrate players	HIGH	Only gate advanced features; core loop always available

Issue	Risk	Mitigation
Unclear how to unlock	MEDIUM	Show clear requirements; hint system; progressive clues
Soft lock if requirements impossible	CRITICAL	Test all unlock paths; provide alternative routes
Pacing feels wrong	MEDIUM	Playtesting with timing metrics; tunable via JSON

Success Criteria

- ✅ Unlocks feel earned, not arbitrary
- ✅ Clear feedback on how to progress
- ✅ No soft locks possible
- ✅ Pacing tested with real players

Phase 7: Polish & Optimization

Duration: 5-7 days **Goal:** Production-ready system

Tasks

1. **Performance Optimization**
    - Profile with Instruments
    - Optimize hot paths (milestone checks)
    - Reduce memory footprint
    - Async loading for story data
2. **Error Handling**
    - Comprehensive error recovery
    - Graceful degradation
    - Clear error messages
    - Telemetry for production issues
3. **Testing**
    - Unit tests for all components
    - Integration tests for full flows
    - Regression tests for save compatibility
    - Stress tests (1000+ journal entries)
4. **Documentation**
    - Code documentation
    - Story authoring guide
    - Architecture overview
    - Debugging guide

5. **Telemetry (Optional)**
    - Anonymous usage metrics
    - Milestone achievement rates
    - Branch distribution
    - Performance metrics

Success Criteria

- ✅ Smooth 60fps with progression active
- ✅ Memory usage acceptable
- ✅ All tests passing
- ✅ Documentation complete
- ✅ Production-ready error handling

Critical Risks & Global Mitigations

1. Save Compatibility

Risk: New progression system breaks existing saves

Mitigation:

- Separate SwiftData container for progression
- Version all models (`@Attribute(.version(1))`)
- Migration strategy for each schema change
- Fallback to empty progression state if load fails
- Never block game load due to progression errors

```
// Safe save loading pattern
func loadGameState() -> GameState {
    let coreState = loadCoreState() // Always succeeds

    do {
        let progressionState = try loadProgressionState()
        return GameState(core: coreState, progression: progressionState)
    } catch {
        logError("Progression load failed: \(error)")
        return GameState(core: coreState, progression: .fresh())
    }
}
```

2. Performance Degradation

Risk: Progression hooks slow down game loop

Mitigation:

- All hooks run async/detached

- Batch journal writes
- Rate-limit milestone checks
- Profile before/after each phase
- Set performance budgets (max 1ms per frame)

3. Thread Safety

Risk: Concurrent access to progression state causes crashes

Mitigation:

- Use `actor` for ProgressionManager (Swift 6)
- Immutable snapshots for condition evaluation
- Copy-on-write for state changes
- Thread Sanitizer in testing

```
actor ProgressionManager {
    private var state: ProgressionState

    func onCommand(input: String, city: City?) async {
        // Atomic state access guaranteed by actor
        let snapshot = state.snapshot()

        // Evaluate conditions on snapshot (thread-safe)
        let triggered = await evaluateMilestones(snapshot, city)

        // Apply changes atomically
        state.apply(triggered)
    }
}
```

4. Story Content Bugs

Risk: Malformed story JSON breaks game

Mitigation:

- Schema validation at build time
- Lint tool for story JSON
- Comprehensive error messages
- Fallback to minimal story
- Never crash due to story errors

5. Complexity Explosion

Risk: System becomes unmaintainable

Mitigation:

- Keep JSON schema simple

- Limit branching depth
- Modular architecture
- Clear separation of concerns
- Regular refactoring

Testing Strategy

Unit Tests

```
// Milestone requirement evaluation
func testRequirements()

// Branch condition checking
func testBranchConditions()

// Journal entry creation
func testJournalLogging()

// Story beat triggering
func testBeatTriggers()

// Save/load persistence
func testPersistence()
```

Integration Tests

```
// Full gameplay flows
func testFirstPlaythrough()

// Branch variations
func testHighTrustPath()
func testLowTrustPath()

// Milestone sequences
func testMilestoneProgression()

// Save/load cycles
func testSaveLoadContinuity()
```

Manual Testing Checklist

- [ ] Play through first 30 minutes
- [ ] Verify all milestones trigger
- [ ] Test each branch condition
- [ ] Confirm story beats display correctly
- [ ] Check save/load preserves state

- ☐ Validate performance (Instruments)
- ☐ Test with existing saves
- ☐ Verify error handling (corrupt JSON)

Rollback Plan

If any phase fails critically:

1. **Immediate Actions**
    - Disable progression hooks (feature flag)
    - Revert to previous commit
    - Document failure mode
2. **Analysis**
    - Identify root cause
    - Assess impact scope
    - Plan fix or redesign
3. **Recovery**
    - Fix issue in isolation
    - Test thoroughly
    - Gradual re-enable
4. **Prevention**
    - Add test coverage for failure case
    - Update documentation
    - Refine next phase plan

Success Metrics

Phase Completion

- All tasks complete
- Success criteria met
- Tests passing
- No regressions
- Performance acceptable

Overall System

- **Stability:** No progression-related crashes
- **Performance:** <1% overhead on game loop
- **Adoption:** 80%+ of sessions have progression active
- **Engagement:** Players complete 50%+ of milestones
- **Satisfaction:** Positive feedback on story/unlocks

Estimated Timeline

Phase	Duration	Cumulative
Phase 0: Foundation	3-5 days	5 days
Phase 1: Observation	5-7 days	12 days
Phase 2: Story Engine	5-7 days	19 days
Phase 3: Milestones	5-7 days	26 days
Phase 4: Branching	7-10 days	36 days
Phase 5: Advanced	7-10 days	46 days
Phase 6: Hard Unlocks	3-5 days	51 days
Phase 7: Polish	5-7 days	58 days

Total: 8-12 weeks (allowing for iteration and testing)

Key Principles

1. **Never break existing functionality**
2. **Test each phase thoroughly before proceeding**
3. **Maintain rollback capability at all times**
4. **Profile performance continuously**
5. **Fail gracefully, never crash**
6. **Keep JSON schema simple and validated**
7. **Document decisions and learnings**

Next Steps

1. Review this plan with stakeholders
2. Set up project tracking (phases as epics)
3. Begin Phase 0 implementation
4. Schedule weekly progress reviews
5. Adjust timeline based on actual velocity

Document Version: 1.0 **Last Updated:** 2025-10-13 **Status:** Ready for Implementation