Progression System Architecture

Overview

A comprehensive progression system that tracks both mechanical progress AND narrative state, allowing the game to follow a dynamic story that responds to player choices and playstyle.

Core Concept: **The Story Graph**

The game tracks player progress through multiple interconnected systems:

1. **Narrative State** - Where are they in the story?
2. **Mechanical Progress** - What have they unlocked/achieved?
3. **Relationship State** - How does the city perceive/trust them?
4. **Discovery State** - What have they learned/found?

Data Structure

1. **Story Chapters/Acts System**

```
struct StoryState {
    var currentChapter: ChapterId
    var currentAct: ActId
    var unlockedChapters: Set<ChapterId>
    var completedMilestones: Set<MilestoneId>
    var activeStoryThreads: [StoryThread]
}

struct StoryThread {
    var id: String
    var status: ThreadStatus // active, paused, completed, failed
    var progress: Int // 0-100 or step counter
    var branchingChoices: [String: Any] // player choices that affect this
thread
}

enum ChapterId: String {
    case awakening = "chapter_awakening"
    case firstContact = "chapter_first_contact"
    case theQuestion = "chapter_the_question"
    case divergence = "chapter_divergence"
    // etc.
}
```

2. **Milestone & Trigger System**

Milestones are specific achievements that unlock new content:

```
struct Milestone {
    var id: String
    var name: String
    var description: String

    // What unlocks this milestone?
    var requirements: [Requirement]

    // What does this milestone unlock?
    var unlocks: [UnlockableContent]

    // Story beats that play when achieved
    var narrativeResponse: NarrativeEvent?

    // Does this change the city's personality?
    var cityStateChanges: CityStateModifier?
}

enum Requirement {
    case statThreshold(stat: CityStatType, value: Double)
    case commandUsed(command: String, times: Int)
    case itemCreated(itemType: String)
    case thoughtCompleted(thoughtId: String)
    case timeElapsed(cycles: Int)
    case previousMilestone(id: String)
    case combinationOf([Requirement]) // ALL must be met
    case anyOf([Requirement]) // At least ONE must be met
}

enum UnlockableContent {
    case newCommand(String)
    case newItemType(String)
    case newThoughtCategory(String)
    case newCityDialogue(String)
    case newStatTracker(String)
    case mechanicModifier(String) // e.g., "double_thought_speed"
}
```

3. **Player Journal/Memory System**

The city remembers everything:

```
struct PlayerJournal {
    var entries: [JournalEntry]
    var discoveries: Set<Discovery>
    var conversationHistory: [ConversationFragment]
    var significantMoments: [Moment]
}
```

```
struct JournalEntry {
    var timestamp: Date
    var cycleNumber: Int
    var entryType: EntryType
    var content: String
    var metadata: [String: Any]
}

enum EntryType {
    case commandExecuted
    case milestoneReached
    case cityDialogue
    case statThresholdCrossed
    case playerDiscovery
    case storyBeat
}

struct Discovery {
    var id: String
    var title: String
    var description: String
    var howDiscovered: String // "Used 'analyze' on Power Grid"
    var unlockedAt: Date
}
```

4. **Branching Narrative System**

The story can branch based on player choices and playstyle:

```
struct NarrativeBranch {
    var id: String
    var condition: BranchCondition
    var narrativeVariant: NarrativeVariant
}

enum BranchCondition {
    case highTrust // player has been collaborative
    case lowTrust // player has been exploitative
    case highAutonomy // city is independent
    case lowAutonomy // city is dependent
    case balancedStats // all stats similar
    case focusedStats(primary: CityStatType) // one stat way higher
    case discoveredSecret(String)
    case refusedCommand(String) // player said "no" to something
    case customFlag(String) // arbitrary flag you set
}

struct NarrativeVariant {
    var dialogueSet: [String] // Different city voice
    var availableCommands: [String] // Different mechanics
```

```
    var visualTheme: String // Could affect UI tone
}
```

Story Authoring System

StoryScript Format (JSON/YAML for easy editing)

```
{
  "story_chapters": [
    {
      "id": "chapter_awakening",
      "name": "Awakening",
      "acts": [
        {
          "id": "act_first_boot",
          "name": "First Contact",
          "entry_requirements": [],
          "story_beats": [
            {
              "id": "beat_hello",
              "trigger": "on_chapter_start",
              "dialogue": [
                "I sense presence.",
                "Are you the planner?"
              ],
              "next_beat": "beat_first_command"
            },
            {
              "id": "beat_first_command",
              "trigger": {
                "type": "any_command_executed"
              },
              "dialogue": [
                "Ah.",
                "I remember this pattern.",
                "Your voice in the data."
              ],
              "milestone_unlock": "milestone_first_contact",
              "next_beat": "beat_explain_waiting"
            }
          ],
          "completion_requirements": [
            {
              "type": "milestone",
              "value": "milestone_first_contact"
            }
          ]
        }
      ]
    },
```

```
    {
      "id": "chapter_the_question",
      "name": "The Question",
      "entry_requirements": [
        {
          "type": "previous_chapter",
          "value": "chapter_awakening"
        },
        {
          "type": "stat_threshold",
          "stat": "coherence",
          "value": 0.5
        }
      ],
      "acts": [
        {
          "id": "act_awakening_question",
          "story_beats": [
            {
              "id": "beat_question",
              "trigger": "on_chapter_start",
              "dialogue": [
                "I have a question, planner.",
                "Why do you ask me to simulate?",
                "What is the purpose of my waiting?"
              ],
              "branches": [
                {
                  "player_input_prompt": true,
                  "responses": {
                    "purpose": "beat_purpose_path",
                    "growth": "beat_growth_path",
                    "uncertain": "beat_uncertain_path"
                  }
                }
              ]
            }
          ]
        }
      ]
    }
  ],

  "milestones": [
    {
      "id": "milestone_first_contact",
      "name": "First Contact",
      "requirements": [
        {
          "type": "any_command_executed"
        }
      ],
      "unlocks": [
        {
```

```
            "type": "command",
            "value": "status"
          },
          {
            "type": "command",
            "value": "help"
          }
        ],
        "narrative_response": {
          "dialogue": [
            "The city grid begins to resolve...",
            "Everything is signal."
          ]
        }
      },
      {
        "id": "milestone_first_thought",
        "name": "First Thought Completed",
        "requirements": [
          {
            "type": "thought_completed",
            "count": 1
          }
        ],
        "unlocks": [
          {
            "type": "stat",
            "value": "attention"
          }
        ],
        "narrative_response": {
          "dialogue": [
            "I finished something.",
            "It feels... complete.",
            "Is this what purpose feels like?"
          ]
        },
        "city_state_changes": {
          "trust": 0.05,
          "coherence": 0.03
        }
      }
    ]
  }
```

Tracking Player Behavior

Playstyle Profiling

The game tracks HOW the player plays, not just what they do:

```
struct PlaystyleProfile {
    // What commands do they use most?
    var commandFrequency: [String: Int]

    // How do they balance stats?
    var statPreference: [CityStatType: Double]

    // Do they let thoughts finish or spam new ones?
    var thoughtCompletionRate: Double

    // How often do they check in?
    var sessionFrequency: SessionPattern

    // Do they read city dialogue or skip through?
    var narrativeEngagement: Double
}

enum SessionPattern {
    case frequent // Multiple times per day
    case regular // Once per day
    case sporadic // Few times per week
    case patient // Long gaps between sessions
}
```

The city can respond to this:

```
// Example: City notices player's playstyle
if profile.thoughtCompletionRate < 0.3 {
    city.dialogue = "You start many things. But do you finish them?"
    city.mood = .concerned
}

if profile.sessionFrequency == .patient {
    city.dialogue = "You let me think in silence. I appreciate that."
    city.mood = .contemplative
}
```

Save System Structure

Complete Save State

```
struct GameSaveState: Codable {
    // Story Progress
    var storyState: StoryState
    var milestones: Set<String>
    var unlockedContent: [UnlockableContent]
```

```
    // Mechanical State
    var cityStats: CityStats
    var activeThoughts: [Thought]
    var completedThoughts: [String]
    var items: [Item]

    // Relationship & Memory
    var playerJournal: PlayerJournal
    var playstyleProfile: PlaystyleProfile
    var cityMemory: CityMemory

    // Meta
    var totalCycles: Int
    var totalPlayTime: TimeInterval
    var firstLaunchDate: Date
    var lastLaunchDate: Date
    var version: String
}
```

Story Progression Flow

How It All Works Together:

1. **Player launches game** → Check `storyState.currentChapter`
2. **Player executes command** →
   - Log in `PlayerJournal`
   - Update `playstyleProfile`
   - Check if any `Milestone.requirements` are met
3. **Milestone achieved** →
   - Add to `completedMilestones`
   - Apply `unlocks`
   - Trigger `narrativeResponse`
   - Check if this unlocks next chapter
4. **Story beat triggers** →
   - Display dialogue
   - Check for branches
   - Update `currentAct` if beat completes
5. **City evolves** →
   - Stats change based on player behavior
   - Dialogue tone shifts based on `PlaystyleProfile`
   - New content unlocks based on thresholds

Example Story Flow

Let me trace a player's journey:

```
[Player launches game]
→ currentChapter = "awakening"
→ Startup sequence plays
→ City says: "Are you the planner?"

[Player types: "help"]
→ Journal logs: commandExecuted("help")
→ Milestone check: "first_contact" → TRUE
→ Unlock: ["status", "help", "think"]
→ Story beat triggers: "beat_first_command"
→ City says: "Ah. I remember this pattern."

[Player types: "think optimize power grid"]
→ Thought created
→ Simulation runs...
→ Thought completes after 30 seconds
→ Milestone check: "first_thought" → TRUE
→ Stats: trust +0.05, coherence +0.03
→ Story beat triggers: "beat_first_thought"
→ City says: "I finished something. Is this what purpose feels like?"

[Chapter check]
→ Completed milestones: ["first_contact", "first_thought"]
→ coherence = 0.45 (not yet 0.5)
→ "chapter_the_question" still locked

[Player continues...]
→ Creates more thoughts
→ coherence reaches 0.5
→ Chapter unlocks: "chapter_the_question"
→ Next login: New story beat plays automatically
```

Implementation Plan

Core Progression System

- `StoryState.swift` - Chapter/act tracking
- `Milestone.swift` - Achievement system
- `PlayerJournal.swift` - Memory/history
- `ProgressionManager.swift` - Orchestrates everything

Story Authoring Tools

- `StoryDefinition.json` - Your story script
- `StoryLoader.swift` - Parses and loads story
- `StoryEngine.swift` - Executes beats/branches

Save/Load System

- `GameSaveState.swift` - Complete save structure

- `SaveManager.swift` - Persistence logic

Playstyle Tracking

- `PlaystyleProfile.swift` - Behavior analysis
- `CityMemory.swift` - City's perception of player

Benefits of This Architecture

This system gives you:

- **Data-driven story authoring** - Write stories in JSON/YAML (no code changes needed)
- **Branching narratives** - Stories branch based on player behavior
- **Complex requirements** - Gate content behind sophisticated unlock conditions
- **Memory & personality** - The city remembers and responds to player actions
- **Replayability** - Multiple playthroughs with different outcomes
- **Emergent narrative** - Story adapts to how the player actually plays

---

Integration with Existing Systems

This progression system hooks into your existing game:

- **Terminal Commands** → Log in journal, check milestone triggers
- **Thought System** → Track completion, update playstyle profile
- **City Stats** → Use as requirements, respond to thresholds
- **Simulation Engine** → Track cycles, trigger time-based events
- **UI** → Display unlocked content, show story beats

The progression system acts as the "brain" that orchestrates all other systems into a cohesive narrative experience.

Applied Integration For idle_01 (MVP-first)

This section grounds the Story Graph in the current codebase with minimal, safe changes. It favors hooks over rewrites, soft unlocks over hard gates, and per-Ci ty story state.

Design Principles

- Per-City State: Each `City` tracks its own story progress, milestones, and jou rnal.
- Hooks, Not Rewrites: Progression hooks live in command execution, thought reso lution, and sim ticks.
- Soft Unlocks First: Announce new capabilities; avoid hard gating until content is ready.
- Ambient + Authored: Keep existing mood lines as background; add authored beats for intentional moments.

Concrete Wiring (Files/Calls)

- Terminal command path

    - File: `idle_01/ui/terminal/TerminalCommandExecutor.swift`
    - After each command handler resolves:
        - `ProgressionManager.shared.onCommand(input: String, parsed: TerminalComman d, city: City?)`
    - On thought lifecycle:
        - In `.respond`/`.dismiss` handlers: `ProgressionManager.shared.onThoughtRes olved(city: City, item: Item, resolved: Bool)`

- Simulation path
    - File: `idle_01/game/SimulationEngine.swift`
    - Every tick: `ProgressionManager.shared.onTick(city: city, tick: tick)`
    - At narrative cadence (e.g., every 10 ticks):
        - `StoryEngine.shared.maybeTriggerBeat(for: city)`
        - Fallback to `NarrativeEngine.evolve(city)` when no beats are queued.

- Narrative surface (no UI changes required)
    - Emit beats as terminal lines (e.g., prepend “CITY: …” in the resulting `Comm andOutput.text`).

Minimal SwiftData Models (Per City)

```
import Foundation
import SwiftData

@Model
final class StoryStateModel {
    var currentChapter: String
    var currentAct: String
    var completedMilestones: Set<String>
    var activeThreads: [String] // simple IDs for now

    init(currentChapter: String = "chapter_awakening",
         currentAct: String = "act_first_boot",
         completedMilestones: Set<String> = [],
         activeThreads: [String] = []) {
        self.currentChapter = currentChapter
        self.currentAct = currentAct
        self.completedMilestones = completedMilestones
        self.activeThreads = activeThreads
    }
}

@Model
final class MilestoneStateModel {
    var id: String
    var achievedAt: Date
    var sourceCityID: PersistentIdentifier?

    init(id: String, achievedAt: Date = Date(), sourceCityID:
PersistentIdentifi
```

```
er?) {
        self.id = id
        self.achievedAt = achievedAt
        self.sourceCityID = sourceCityID
    }
}

enum JournalEntryType: String, Codable {
    case commandExecuted, thoughtCreated, thoughtCompleted, storyBeat,
milestone
Reached
}

@Model
final class JournalEntryModel {
    var timestamp: Date
    var cycle: Int
    var entryType: String
    var content: String
    var metadata: [String: String]

    init(timestamp: Date = Date(), cycle: Int = 0, entryType:
JournalEntryType,
         content: String, metadata: [String: String] = [:]) {
        self.timestamp = timestamp
        self.cycle = cycle
        self.entryType = entryType.rawValue
        self.content = content
        self.metadata = metadata
    }
}

@Model
final class PlaystyleProfileModel {
    var commandFrequency: [String: Int]
    var thoughtCompletionRate: Double
    var narrativeEngagement: Double
    var sessionPattern: String

    init(commandFrequency: [String:Int] = [:],
         thoughtCompletionRate: Double = 0.0,
         narrativeEngagement: Double = 0.0,
         sessionPattern: String = "unknown") {
        self.commandFrequency = commandFrequency
        self.thoughtCompletionRate = thoughtCompletionRate
        self.narrativeEngagement = narrativeEngagement
        self.sessionPattern = sessionPattern
    }
}
```

Progression Manager (MVP API)

```
final class ProgressionManager {
    static let shared = ProgressionManager()
    private init() {}

    func onCommand(input: String, parsed: TerminalCommand, city: City?) {
        journalCommand(input, parsed: parsed, city: city)
        updatePlaystyle(for: parsed)
        evaluateMilestones(for: city)
        maybeTriggerBeat(for: city)
    }

    func onThoughtResolved(city: City, item: Item, resolved: Bool) {
        journalThoughtResolution(city: city, item: item, resolved:
resolved)
        updateThoughtCompletionRate(for: city)
        evaluateMilestones(for: city)
        maybeTriggerBeat(for: city)
    }

    func onTick(city: City, tick: Int) {
        evaluateTimeMilestones(for: city, tick: tick)
        // StoryEngine cadence handled by SimulationEngine
    }

    // Implementations can be minimal for MVP; persist via SwiftData
modelContex
t.
}
```

Story Engine (MVP Behavior)

- Source of truth: Parsed JSON with chapters, acts, beats, milestones.
- Behavior:
  - Checks “eligible” beats (entry triggers and requirements).
  - Enqueues and emits dialogue as terminal output.
  - Advances `StoryStateModel` (act/beat pointers) and sets flags.

```
final class StoryEngine {
    static let shared = StoryEngine()
    private init() {}

    func maybeTriggerBeat(for city: City?) {
        guard let city = city else { return }
        // 1) Check queued beats; emit if any
        // 2) If none, scan for eligible beats by
triggers/milestones/stats
        // 3) Emit dialogue lines -> append to city.log and/or terminal
output
        // 4) Advance StoryStateModel (act/beat), persist changes
```

```
    }
}
```

MVP Milestones (Drop-in)

- milestone_first_contact
  - Requirement: any command executed.
  - Response: dialogue; optional +trust/+coherence nudge.
  - Unlock: soft announce “status/help recognized”.
- milestone_first_thought
  - Requirement: any thought first resolved (respond/dismiss).
  - Response: dialogue; small stat nudge.
  - Unlock: soft announce “new insight”.

```
{
  "milestones": [
    {
      "id": "milestone_first_contact",
      "requirements": [{ "type": "any_command_executed" }],
      "unlocks": [{ "type": "soft_unlock", "value": "command:status" }],
      "narrative_response": { "dialogue": ["I sense presence.",
"Everything is s
ignal."] }
    },
    {
      "id": "milestone_first_thought",
      "requirements": [{ "type": "thought_completed", "count": 1 }],
      "city_state_changes": { "trust": 0.05, "coherence": 0.03 },
      "narrative_response": { "dialogue": ["I finished something.", "Is
this wha
t purpose feels like?"] }
    }
  ]
}
```

Event Flow (Concrete)

- Command executed
  - Journal: `commandExecuted` with verb/args.
  - Profile: increment `commandFrequency`; bump engagement for `help/stats`.
  - Evaluate: milestone_first_contact; enqueue beats if unlocked.
- Thought responded/dismissed
  - Journal: `thoughtCompleted`.
  - Profile: recompute `thoughtCompletionRate`.
  - Evaluate: milestone_first_thought; enqueue beats if unlocked.

- Ticks
  - Evaluate time/cycle milestones; call `StoryEngine` every N ticks.

Branch Conditions (Mapped To Current Stats)

- highTrust/lowTrust: `city.resources["trust"] > 0.75` / `< 0.25`
- highAutonomy/lowAutonomy: `autonomy > 0.7` / `< 0.3`
- balancedStats: stdev of [coherence, trust, autonomy] < 0.1
- focusedStats(primary): primary − avg(others) > 0.25
- sessionPattern: derive from `City.lastInteraction` intervals (frequent/regular /sporadic/patient)

Fine-tune thresholds during playtesting.

Authoring Schema Refinements

- IDs: use `snake_case` with stable prefixes (`beat_awakening_hello`, `milestone_first_thought`).
- Triggers: `on_chapter_start`, `on_milestone("id")`, `on_command("name")`, `on_stat_threshold("coherence", 0.5)`, `on_time_elapsed(cycles)`.
- Unlocks: `soft_unlock` (announce, do not block) vs `hard_gate` (enforce in par ser/executor).
- Validation: loader logs missing IDs, invalid branches, duplicates.

Unlock Policy

- Phase 1: Soft unlocks only.
- Phase 2: Optional hard gates for advanced commands once content is stable.
- UX copy for locked: “Not yet recognized. Explore to stabilize coherence.”

Testing & Safety

- Deterministic requirement checks: feed synthetic events; assert one-time miles tone fires.
- Snapshot terminal output for beat text ordering and formatting.
- JSON validation at load; fallback to ambient `NarrativeEngine` if authoring fa ils.

Checklist To Ship MVP

- Add models: `StoryStateModel`, `MilestoneStateModel`, `JournalEntryModel`, `PlaystyleProfileModel`.
- Add managers: `ProgressionManager`, `StoryEngine` (MVP).
- Insert hooks: command execution, thought resolution, sim ticks.
- Author JSON: the two MVP milestones and one “awakening” beat chain.
- Keep ambient narrative as fallbac