Balatro AI Assistant: Technical Feasibility and Architecture Outline

Persona: You are a Senior Solutions Architect and Technical Product Manager. Your task is to analyze the following application concept, determine its primary technical challenges and dependencies, and outline the high-level architecture required to achieve the goals.

The Application: Balatro AI Coach

Goal: To create a real-time, PC-based desktop application that provides dynamic, adaptive tactical guidance for the roguelike deckbuilding game Balatro based on optimal scoring strategies, Joker synergies, and meta-game patterns.

Platform: PC (Windows/Mac/Linux) using Node.js runtime **Implementation Scope:** Local desktop application with screen capture and overlay capabilities

---

1. Core Functionality & Real-Time Analysis

Feature 1: Real-Time Visual/Game State Processing (Screen Analysis)
Description: The app must run as an overlay or companion process and analyze the Balatro game window in real-time. It must identify:

1. **Current Hand (8 cards)** - Rank, suit, and any enhancements (Bonus, Mult, Wild, Glass, Steel, Stone, Gold, Lucky)
2. **Card Seals** - Red Seal (retrigger), Gold Seal ($3 per round), Blue Seal (creates Planet card), Purple Seal (creates Tarot)
3. **Active Jokers (up to 5 slots)** - Type, edition (Foil +50 chips, Holographic +10 mult, Polychrome x1.5 mult), special tags (Eternal, Perishable, Rental)
4. **Poker Hand Levels** - Current level for each of the 12 poker hands
5. **Current Blind Info** - Small/Big/Boss, target chip requirement, modifiers, remaining hands/discards
6. **Shop State** - Available Jokers (rarity: Common, Uncommon, Rare), Tarot cards, Planet cards, Vouchers, current money
7. **Deck State** - Total cards in deck, consumable slots (2 base), money, ante number
8. **Scoring Calculation** - Real-time score prediction based on selected cards

Technical Requirements:

```
javascript

// Core requirements
- Screen capture: 30-60 FPS window capture (using node-screenshot or Electron's desktopCapturer)
- OCR for text: Tesseract.js for reading card values, money, scores, hand levels
- Computer Vision: TensorFlow.js or ONNX Runtime for object detection
- Image processing: Sharp or Jimp for preprocessing
- Low-latency: <200ms total pipeline (capture → process → analyze → display)
```

Key Challenges:

- Balatro uses pixel art with specific color palettes - easier for CV than photorealistic games
- Multiple card enhancements create visual variations (need training data for each combination)
- Boss Blinds can flip cards face-down (???) - must handle uncertainty
- Shop rerolls change layout dynamically

Feature 2: Tactical Guidance (Real-Time Recommendations)

Description: Based on current game state, the AI must provide optimal recommendations for:

During Blind (Playing Phase):

- **Best hand to play** - Which 5 cards maximize score given current Jokers
- **Cards to discard** - Which cards to discard to improve hand options
- **Play vs Discard decision** - When to commit vs when to fish for better hands
- **Card order optimization** - Left-to-right placement matters for triggers (Glass cards, Mult cards, Bloodstone, etc.)

In Shop (Between Blinds):

- **Joker purchases** - Which Joker best synergizes with current build
- **Consumable priorities** - Buy Tarot (deck modification) vs Planet (hand upgrades) vs reroll
- **Deck thinning strategy** - Use Death card to copy best card, Hanged Man to destroy cards
- **Economy decisions** - Save for interest ($5 per $25 held, max $25 interest cap) vs immediate purchases
- **Blind skip decisions** - Skip Small Blind for tags (free Uncommon Joker, money, booster packs)

Technical Requirements:

```
javascript
```

```
// Decision Engine Components
- Scoring Engine: Implement full Balatro score calculation
  * Score = (Base Chips + Card Chips) × (Base Mult + Extra Mult) × XMult modifiers
  * Must account for: hand type, card order, Joker effects, card enhancements

- Strategy Evaluator:
  * Hand strength calculator (simulates all possible 5-card combinations)
  * Synergy scorer (rates Joker compatibility with current deck)
  * Expected value calculator (probabilistic future hand analysis)

- Build Classifier:
  * Identifies current build archetype (Flush, Straight, Mult-stack, XMult scaling, etc.)
  * Recommends Jokers/Tarots that strengthen current path
```

Output Display:

- Semi-transparent overlay window with:
  - Recommended play (highlighted cards)
  - Expected score if played
  - Alternative plays ranked by score
  - Shop recommendations with reasoning
  - Build strength indicator (traffic light system)

Feature 3: Mid-Game Activation & State Estimation

Description: The app must be able to start at any point during a run and:

1. Detect current ante (1-8+)
2. Identify all active Jokers and their properties
3. Estimate deck composition (if not all cards visible)
4. Enter analysis loop within 1-2 seconds

Technical Requirements:

```
javascript
```

```
// Robust initial state detection
- Use OCR to read ante number (top right)
- Capture all visible Jokers in slots
- Detect current money for economy tracking
- Analyze visible cards in hand for enhancement patterns
- Query user for unknown state (interactive calibration)
```

Challenge:

- Cannot see full deck without playing through it
- Must make probabilistic assumptions about unseen cards
- Boss Blind effects may not be immediately obvious

---

2. AI & Strategy Knowledge System (The "Meta Engine")

Requirement 1: Strategy Database & Pattern Recognition

Description: The system must contain comprehensive knowledge of:

- All 150+ Jokers and their synergies
- 12 poker hand types and optimal upgrade paths
- 22 Tarot cards and strategic use cases
- 14 Planet cards and hand-type focus recommendations
- 11 Spectral cards (rare, high-risk/reward)
- Boss Blind counters and adaptation strategies

Data Strategy:

```javascript
```

```
// Knowledge Base Structure (JSON/SQLite)
{
 "jokers": [
   {
    "id": "blueprint",
    "name": "Blueprint",
    "rarity": "rare",
    "effect": "Copies ability of Joker to the right",
    "synergies": ["baron", "cavendish", "brainstorm"],
    "antisynergies": ["showman", "troubadour"],
    "build_types": ["xmult", "scaling", "combo"],
    "priority_score": 9.5
   }
  // ... 149 more jokers
 ],

 "strategies": [
   {
    "name": "All Flushes",
    "description": "Focus on single suit, build around flush synergies",
    "key_jokers": ["smeared_joker", "sock_and_buskin", "luchador"],
    "planet_focus": "Neptune (Flushes)",
    "tarot_priorities": ["strength", "lovers", "death"],
    "difficulty": "easy",
    "scaling_type": "linear → quadratic with upgrades"
   }
  // ... more strategies
 ]
}
```

Learning Sources: Since we're building locally and not scraping live content, the knowledge base will be:

1. **Manually curated** from Balatro Wiki data
2. **Updated from community guides** (PCGamer, Reddit, Steam guides)
3. **Simulation-based optimization** (run Monte Carlo simulations of builds)

Requirement 2: Context-Aware Strategy Recommendations

Description: The AI must understand:

- **Current build state** - What's the core strategy? (Flush focus, Mult stacking, Planet spam, etc.)
- **Pivot opportunities** - When a better strategy becomes available (e.g., finding Blueprint + Baron)

- **Ante-appropriate scaling** - Early game (Antes 1-2) vs Mid (3-6) vs Late (7-8)
- **Boss preparation** - Adjust strategy before Boss Blinds (e.g., avoid suit-specific builds against "The Arm")

Example Decision Trees:

javascript

```
// Ante 1-2: Early Game Priorities
if (currentMoney < 25 && !hasEconomyJoker) {
  recommendJokers = filterByType("economy"); // Bull, Egg, Space Joker
} else if (!hasChipSource && !hasMultSource) {
  recommendJokers = filterByEffect("additive_scaling"); // Scary Face, Smiley Face, Fortune Teller
}

// Ante 3-6: Mid Game Pivot
if (hasClearBuildPath) {
  recommendJokers = findSynergies(currentJokers); // Double down on current strategy
  recommendPlanets = focusOnPrimaryHand();
} else {
  recommendJokers = filterByVersatility("high"); // Abstract, Joker, Supernova
}

// Ante 7-8: Late Game XMult Required
if (xMultCount < 1) {
  warningLevel = "CRITICAL";
  recommendJokers = filterByType("xmult"); // Baron, Bloodstone, Cavendish
}
```

Requirement 3: Adaptive Strategy Engine (User Deviation Handling)

Description: If the user ignores the AI's suggestion and makes a different play, the AI must:

1. **Evaluate the user's unprompted move** - Was it suboptimal, or did user see something AI missed?
2. **Infer the user's new tactical plan** - Did they pivot to a different build?
3. **Adjust subsequent suggestions** - Support user's chosen direction rather than forcing original plan

Implementation:

javascript

```
// Intent Inference System
class StrategyTracker {
  constructor() {
    this.predictedActions = [];
    this.actualActions = [];
    this.inferredStrategy = null;
  }

  recordDeviation(predicted, actual) {
    if (predicted.joker !== actual.joker) {
      // User bought different Joker - analyze why
      const userChoice = analyzeJokerIntent(actual.joker, currentBuild);

      if (userChoice.confidence > 0.7) {
        this.inferredStrategy = userChoice.buildPath;
        console.log(`User pivoting to: ${userChoice.buildPath}`);
      }
    }
  }

  adjustRecommendations() {
    if (this.inferredStrategy) {
      return getJokersForStrategy(this.inferredStrategy);
    }
    return getDefaultRecommendations();
  }
}
```

Key Insight: This is simpler than real-time game opponents because Balatro is single-player PvE. The AI just needs to recognize when the user is building toward a different synergy and align with it.

---

3. Technical & Deployment Requirements

Platform: PC Desktop (Node.js + Electron)

Why Electron:

- Cross-platform (Windows, Mac, Linux)
- Native window capture APIs
- Overlay rendering capabilities
- NPM ecosystem for ML libraries

- Easy packaging/distribution

Tech Stack:

```javascript
// Core Framework
- Electron 28+ (desktop app framework)
- Node.js 20+ (runtime)
- TypeScript (type safety for complex game logic)

// Screen Capture & Processing
- @ffmpeg/ffmpeg (video processing)
- node-screenshots (native screen capture)
- sharp (image processing)
- tesseract.js (OCR)

// Machine Learning
- TensorFlow.js (object detection)
- ONNX Runtime (if using pre-trained models)

// UI/Overlay
- React (UI components)
- Electron's BrowserWindow with transparency
- HTML5 Canvas for visual indicators

// Data & Storage
- SQLite (local strategy database)
- LocalStorage (user preferences)
```

Cross-Platform Compatibility

Platform	Implementation Status	Notes
Windows	✅ Priority 1	Full support for screen capture (GetWindowDC API)
Mac	✅ Priority 2	Requires accessibility permissions for screen recording
Linux	⚠️ Priority 3	X11/Wayland support varies, may need fallback methods

Distribution:

- Self-contained Electron app (.exe / .dmg / .AppImage)
- No dependency on app stores
- Easy updates via Electron Updater

- Local-only (no cloud requirements)

User Experience (UX)

Activation Flow:

1. Launch Balatro AI Assistant
2. Select Balatro game window from list
3. Click "Start Analysis"
4. Overlay appears with recommendations

In-Game Controls:

- **Hotkey Toggle** - Show/hide overlay (default: F9)
- **Pause Analysis** - Temporarily stop recommendations (F10)
- **Settings Panel** - Adjust overlay opacity, recommendation verbosity
- **Manual Calibration** - If detection fails, user can input current state

Performance:

- Idle CPU: <2% (monitoring for game window)
- Active CPU: 10-15% (during analysis)
- RAM: ~150-200MB
- Minimal impact on game performance

---

4. AI Output Requirements

Target AI Deliverables:

1. Top 3 Greatest Technical Risks:

Risk #1: Screen Capture Reliability & Game Window Detection ⚠️ HIGH

Challenge:

- Balatro can run in fullscreen, windowed, or borderless modes
- Different resolutions (1080p, 1440p, 4K) change pixel positions
- Steam Overlay and Discord Overlay can interfere
- Electron's screen capture may have permission issues on Mac

Mitigation:

- Implement template matching for UI elements (anchor detection)
- Support multiple resolution profiles
- Fallback to manual region selection if auto-detection fails
- Test extensively on all three platforms

Risk #2: Real-Time Performance vs Accuracy Trade-off ⚠️ MEDIUM

Challenge:

- Running CV models on every frame (30-60 FPS) is computationally expensive
- Users expect <200ms recommendation latency
- Balatro games are fast-paced (3-5 second decision windows during blinds)

Mitigation:

- Adaptive frame rate: Only analyze when game state changes (detect motion/UI changes)
- Use lightweight models (MobileNet, YOLO-Nano)
- Implement caching: Don't reanalyze identical frames
- Progressive enhancement: Basic recommendations fast, detailed analysis async

Risk #3: Comprehensive Strategy Knowledge Maintenance ⚠️ MEDIUM

Challenge:

- Balatro receives frequent updates (new Jokers, balance changes)
- 150+ Jokers with complex interactions require continuous testing
- User-discovered synergies may not be in initial knowledge base

Mitigation:

- JSON-based knowledge files (easy to update without recompiling)
- Community contribution system (users can submit strategy updates)
- Automated testing suite for scoring calculations
- Version detection (warn if Balatro version doesn't match knowledge base)

---

2. High-Level Three-Part Architecture:

Part A: Screen Analysis Layer (Perception Module)

```
┌─────────────────────────────────────────────────────┐
```

```
│   SCREEN ANALYSIS LAYER (Node.js)    │
├──────────────────────────────────────────────────┤
│                        │
│  ┌──────────────────────────────────────────┐  │
│  │ Window Capture Service            │  │
│  │ - Target Balatro game window      │  │
│  │ - 30 FPS screenshot stream        │  │
│  │ - Motion detection (smart FPS)    │  │
│  └────────────────┬─────────────────────────┘  │
│         │                │
│         ▼                │
│  ┌──────────────────────────────────────────┐  │
│  │ Image Preprocessing Pipeline      │  │
│  │ - Resize/normalize                │  │
│  │ - Region of Interest extraction   │  │
│  │ - Color quantization              │  │
│  └────────────────┬─────────────────────────┘  │
│         │                │
│         ▼                │
│  ┌──────────────────────────────────────────┐  │
│  │ Computer Vision Models            │  │
│  │ - Card Detection (TF.js YOLO)     │  │
│  │ - Text Recognition (Tesseract)    │  │
│  │ - UI Element Classification       │  │
│  └────────────────┬─────────────────────────┘  │
│         │                │
│         ▼                │
│  ┌──────────────────────────────────────────┐  │
│  │ Game State Parser                 │  │
│  │ - Current hand (8 cards)          │  │
│  │ - Active Jokers (5 slots)         │  │
│  │ - Blind info (target, hands left)│  │
│  │ - Shop state (purchases available│  │
│  │ - Money, ante, deck size          │  │
│  └──────────────────────────────────────────┘  │
└─────────────────┬────────────────────────────────┘
        │ (emits GameState object)
        ▼
```

Technologies:

- `electron-screenshots` - Window capture
- `sharp` - Image processing

- `@tensorflow/tfjs-node` - Object detection
- `tesseract.js` - OCR for text
- Custom parsers for game-specific UI

Part B: Decision Engine (Strategy Module)

```
┌─────────────────────────────────────────┐
│    DECISION ENGINE (Node.js Core)       │
├─────────────────────────────────────────┤
│                                         │
│  ┌───────────────────────────────────┐  │
│  │ Knowledge Base (SQLite)           │  │
│  │ - Joker database (150+)           │  │
│  │ - Strategy patterns               │  │
│  │ - Synergy matrix                  │  │
│  │ - Boss Blind counters             │  │
│  └───────────────┬───────────────────┘  │
│                  │                      │
│                  ▼                      │
│  ┌───────────────────────────────────┐  │
│  │ Scoring Engine                    │  │
│  │ - Full Balatro calculation        │  │
│  │ - (Chips) × (Mult) × (XMult)      │  │
│  │ - Card order optimization         │  │
│  │ - Simulates all 5-card combos     │  │
│  └───────────────┬───────────────────┘  │
│                  │                      │
│                  ▼                      │
│  ┌───────────────────────────────────┐  │
│  │ Strategy Analyzer                 │  │
│  │ - Build classifier                │  │
│  │ - Pivot detection                 │  │
│  │ - Joker synergy scorer            │  │
│  │ - Economy optimizer               │  │
│  └───────────────┬───────────────────┘  │
│                  │                      │
│                  ▼                      │
│  ┌───────────────────────────────────┐  │
│  │ Recommendation Engine             │  │
│  │ - Best hand to play               │  │
│  │ - Cards to discard                │  │
│  │ - Shop purchase priority          │  │
│  │ - Skip/play blind decision    │ │
│  └───────────────────────────────┘ │
└──────────────┬─────────────────────┘
               │ (emits Recommendations)
               ▼
```

Technologies:

- `better-sqlite3` - Fast local database
- Custom scoring algorithms (TypeScript)
- Decision tree for strategy selection
- Monte Carlo simulation for probabilistic plays

Part C: User Interface Layer (Presentation Module)

```
┌────────────────────────────────────┐
│  USER INTERFACE (Electron + React)  │
├────────────────────────────────────┤
│                 │                  │
│  ┌───────────────────────────────┐ │
│  │ Transparent Overlay Window    │ │
│  │ - Always-on-top               │ │
│  │ - Click-through mode          │ │
│  │ - Draggable/resizable         │ │
│  └──────────┬────────────────────┘ │
│       │             │              │
│       ▼             │              │
│  ┌───────────────────────────────┐ │
│  │ Recommendation Display        │ │
│  │ ┌─────────────────────────┐   │ │
│  │ │ BEST PLAY: 9♥ J♥ Q♥ K♥ A♥ │ │ │
│  │ │ Hand: Straight Flush    │ │ │
│  │ │ Score: 2,456,789        │ │ │
│  │ └─────────────────────────┘   │ │
│  │ ┌─────────────────────────┐   │ │
│  │ │ SHOP: Buy "Blueprint" (Rare)│ │ │
│  │ │ Synergy Score: 9.5/10   │ │ │
│  │ │ Reason: Copies Baron (+mult)│ │ │
│  │ └─────────────────────────┘   │ │
│  └───────────────────────────────┘ │
│                 │                  │
│  ┌───────────────────────────────┐ │
│  │ Control Panel (Separate Window)  │ │
│  │ - Start/Stop analysis            │ │
│  │ - Opacity slider                 │ │
│  │ - Recommendation detail level    │ │
│  │ - Manual state correction        │ │
│  └──────────────────────────────────┘ │
└───────────────────────────────────────┘
```

Technologies:

- `React 18` - UI components
- `Electron's BrowserWindow` - Overlay
- `CSS Grid/Flexbox` - Layout
- `HTML5 Canvas` - Optional visual indicators

---

3. Key ML Models Required:

For Part A (Screen Analysis):
Model 1: Card Detection & Classification

- **Architecture:** YOLOv8-nano (real-time object detection)
- **Input:** 1920×1080 screenshot (scaled to 640×640)
- **Output:** Bounding boxes + class labels
- **Classes (~200 total):**
  - 52 base cards (13 ranks × 4 suits)
  - Card enhancements (Bonus, Mult, Wild, Glass, Steel, Stone, Gold, Lucky)
  - Card seals (Red, Gold, Blue, Purple)
  - Card editions (Foil, Holographic, Polychrome)
- **Training Data Required:** 5,000+ labeled screenshots
- **Optimization:** Quantize to INT8, target 50ms inference on CPU

Model 2: Joker Recognition

- **Architecture:** ResNet-18 (image classification)
- **Input:** Cropped Joker card region (256×256)
- **Output:** Joker ID (1 of 150+)

- **Challenge:** Similar visual styles, must differentiate via details
- **Training Data Required:** 100+ examples per Joker
- **Optimization:** Quantized model, <20ms inference

Model 3: Text Recognition (OCR)

- **Architecture:** Tesseract.js (pre-trained)
- **Input:** Cropped text regions (money, scores, hand levels)
- **Output:** String
- **Preprocessing:** Threshold, denoise, scale up 2x for clarity
- **Target Accuracy:** >95% for numbers

For Part B (Strategy Engine):
Model 4: Build Classifier

- **Architecture:** Random Forest Classifier
- **Input:** Feature vector (current Jokers, deck composition, hand levels)
- **Output:** Build type (Flush, Straight, High Card spam, Mult-stack, XMult scaling, etc.)
- **Training Data:** Simulated + real game data
- **Purpose:** Identify current strategy to recommend coherent purchases

Model 5: Synergy Scorer

- **Architecture:** Graph Neural Network (optional) or Rule-Based System
- **Input:** Current Jokers + proposed new Joker
- **Output:** Compatibility score (0-10)
- **Implementation:** Precomputed synergy matrix + dynamic bonuses
- **Example:**
  - Blueprint + Baron = 10/10 (Blueprint copies Baron's XMult)
  - Blueprint + Egg = 3/10 (no synergy)

Model 6: Monte Carlo Simulator (Not ML, but critical)

- **Purpose:** Simulate 1,000+ possible plays to find optimal discard strategy
- **Algorithm:**
  1. For each possible discard action

   2. Simulate drawing new cards (probabilistic)
   3. Calculate expected score of best possible hand
   4. Rank discard options by expected value

- **Performance:** Must run in <100ms

---

4. Feasibility Score for Low-Latency Implementation:

Overall Feasibility: 7.5/10 ✅ VIABLE

Breakdown by Component:

Component	Feasibility	Reasoning
Screen Capture	9/10	Electron has robust APIs, Balatro is 2D (easier than 3D games)
Card Detection CV	7/10	Pixel art is easier to detect, but enhancements multiply complexity
OCR (Text)	8/10	Fixed-width fonts, high contrast - good for OCR
Strategy Database	9/10	Static knowledge base, manually curated from Wiki
Scoring Engine	8/10	Complex but deterministic, no AI needed
Real-Time Performance	6/10	200ms latency budget is tight for CPU-bound CV
Cross-Platform	7/10	Windows easy, Mac requires permissions, Linux variable
User Experience	8/10	Overlay UX is well-understood, Electron makes this straightforward

Why 7.5/10 instead of higher:

- Real-time CV on CPU (no GPU requirement) is the bottleneck
- Comprehensive Joker synergy knowledge requires significant manual curation
- Balatro updates frequently (new cards break knowledge base)
- Some Joker interactions are complex (order-dependent, probabilistic)

Why not lower:

- Balatro is turn-based, not frame-perfect (users have 5-10 seconds to decide)
- 2D pixel art is CV-friendly (unlike photorealistic 3D games)
- No server/cloud needed (all local processing)
- No legal concerns (single-player game, no competitive advantage)

---

5. Implementation Roadmap (If Proceeding)

Phase 1: Proof of Concept (2-4 weeks)

Goals:

1. Basic screen capture of Balatro window
2. Detect hand cards (rank/suit only, no enhancements yet)
3. Simple scoring engine (calculate score for one hand type)
4. Overlay with text recommendations

Deliverables:

- Electron app that captures Balatro window
- Basic card detection (using template matching, not ML yet)
- Hardcoded strategy: "Always play highest poker hand"
- Text overlay showing recommendation

Risk Reduction:

- Validates screen capture works on target platforms
- Tests overlay rendering performance
- Proves scoring logic is implementable

Phase 2: Core ML & Strategy (6-8 weeks)

Goals:

1. Train YOLO model for full card detection (enhancements, seals)
2. Implement Joker recognition (ResNet classifier)
3. Build comprehensive scoring engine (all poker hands, modifiers)
4. Create knowledge base (150+ Jokers, synergies)
5. Shop recommendation system

Deliverables:

- Trained CV models with >90% accuracy
- Full score calculator matching Balatro exactly
- SQLite database with Joker data

- Shop purchase recommendations

Risk Reduction:

- Proves CV accuracy is sufficient
- Validates strategy logic against real gameplay

Phase 3: Adaptive Intelligence (4-6 weeks)

Goals:

1. Build classifier (identify Flush build vs Straight build, etc.)
2. Intent inference (detect user pivots)
3. Dynamic recommendation adjustment
4. Boss Blind counters

Deliverables:

- AI that adapts to user's chosen strategy
- Boss Blind preparation warnings
- Pivot suggestions ("You found Blueprint, consider XMult build")

Phase 4: Polish & Distribution (3-4 weeks)

Goals:

1. Performance optimization (reduce CPU/RAM usage)
2. Cross-platform testing (Windows/Mac/Linux)
3. User settings (overlay opacity, hotkeys, recommendation detail)
4. Installer/updater
5. Documentation

Deliverables:

- Packaged Electron app (.exe, .dmg, .AppImage)
- User guide and tutorial
- GitHub repository (if open-source)

---

6. Alternative Approaches

Option A: Simplified "Companion App" (No Screen Capture)

Description: Instead of screen capture, user manually inputs game state via UI:

- Select 8 cards in hand
- Check boxes for active Jokers
- Enter money, ante, blind target

Pros:

- No CV required (eliminates Risk #1 and #2)
- 100% accuracy (no detection errors)
- Runs on any platform (even mobile/web)
- Much faster to build (2-4 weeks vs 3-6 months)

Cons:

- Manual input is tedious (breaks game flow)
- Slower than automated analysis
- Less "magic" factor

Verdict: Good for MVP/testing, but defeats the "real-time" goal

---

Option B: Post-Game Analysis Tool

Description: User uploads Balatro game log or screenshot at end of run:

- AI analyzes decisions made
- Points out missed opportunities
- Suggests build improvements

Pros:

- No real-time pressure (can use more complex ML)
- Educational value (learn from mistakes)
- No overlay/screen capture needed

Cons:

- Not useful during active gameplay

- Balatro doesn't export game logs (would need manual screenshots)

Verdict: Complementary feature, not replacement for real-time guidance

---

7. Final Recommendations

Should You Build This?

YES, with caveats:

✅ **Pros:**

- Technically feasible with current technology
- Balatro is a perfect candidate (2D, turn-based, single-player)
- No legal/ethical concerns (unlike competitive games)
- Large potential user base (Balatro sold 3.5M+ copies)
- Educational value (teaches players optimal strategy)

⚠️ **Cons:**

- 3-6 months development time for full version
- Requires computer vision expertise (training models)
- Maintenance burden (Balatro updates frequently)
- Performance optimization is non-trivial

Recommended Path:

Start with Phase 1 POC (2-4 weeks):

- Validate screen capture works
- Build basic scoring engine
- Test overlay UX
- If successful → proceed to Phase 2
- If blocked by performance/accuracy → pivot to Option A (Companion App)

Key Success Criteria:

- Card detection >90% accuracy
- Recommendation latency <200ms
- CPU usage <20% during active analysis

- User feedback positive on UX

Estimated Total Development Cost:

Solo Developer (experienced): 3-6 months full-time **Small Team (2-3 developers):** 2-4 months **Budget (if outsourcing):** $30K-$60K

---

8. Appendix: Sample Code Structures

A. Game State Object (TypeScript)

```
typescript
```

```
interface Card {
  rank: '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '10' | 'J' | 'Q' | 'K' | 'A';
  suit: 'Spades' | 'Hearts' | 'Clubs' | 'Diamonds';
  enhancement?: 'Bonus' | 'Mult' | 'Wild' | 'Glass' | 'Steel' | 'Stone' | 'Gold' | 'Lucky';
  seal?: 'Red' | 'Gold' | 'Blue' | 'Purple';
  edition?: 'Foil' | 'Holographic' | 'Polychrome';
}

interface Joker {
  id: string;
  name: string;
  rarity: 'Common' | 'Uncommon' | 'Rare' | 'Legendary';
  edition?: 'Foil' | 'Holographic' | 'Polychrome';
  eternal?: boolean;
  perishable?: boolean;
  rental?: boolean;
}

interface GameState {
  // Current Play Area
  hand: Card[]; // 8 cards
  selectedCards: Card[]; // 0-5 cards player has selected

  // Jokers
  jokers: Joker[]; // 0-5 active jokers

  // Blind Info
  blindType: 'Small' | 'Big' | 'Boss';
  blindName?: string; // e.g., "The Hook", "The Eye"
  targetScore: number;
  handsRemaining: number;
  discardsRemaining: number;

  // Deck State
  deckSize: number;
  consumableSlots: number;

  // Economy
  money: number;
  ante: number; // 1-8+

  // Hand Levels
  handLevels: {
    'High Card': number;
    'Pair': number;
    'Two Pair': number;
    'Three of a Kind': number;
    'Straight': number;
    'Flush': number;
    'Full House': number;
    'Four of a Kind': number;
    'Straight Flush': number;
    'Flush Five': number;
    'Flush House': number;
    'Five of a Kind': number;
  };

  // Shop State (if in shop)
  shopJokers?: Joker[];
  shopTarots?: string[];
  shopPlanets?: string[];
  shopVouchers?: string[];
  rerollCost: number;
}
```

B. Scoring Engine (TypeScript)

typescript

```
class ScoringEngine {
  calculateScore(hand: Card[], gameState: GameState): number {
    // Step 1: Identify poker hand type
    const handType = this.identifyHandType(hand);

    // Step 2: Get base chips and mult from hand level
    const baseChips = this.getBaseChips(handType, gameState.handLevels[handType]);
    const baseMult = this.getBaseMult(handType, gameState.handLevels[handType]);

    // Step 3: Add card chips
    let chips = baseChips;
    for (const card of hand) {
      chips += this.getCardChips(card);
    }

    // Step 4: Calculate mult (additive)
    let mult = baseMult;

    // Add mult from Mult cards
    for (const card of hand) {
      if (card.enhancement === 'Mult') mult += 4;
    }

    // Add mult from Jokers (order matters!)
    for (const joker of gameState.jokers) {
      mult += this.getJokerMult(joker, hand, gameState);
    }

    // Step 5: Apply multiplicative mult (XMult)
    let xMult = 1.0;

    // Glass cards
    for (const card of hand) {
      if (card.enhancement === 'Glass') xMult *= 2.0;
    }

    // XMult Jokers
    for (const joker of gameState.jokers) {
      xMult *= this.getJokerXMult(joker, hand, gameState);
    }

    // Polychrome edition
    for (const card of hand) {
      if (card.edition === 'Polychrome') xMult *= 1.5;
    }

    // Final calculation
    return Math.floor(chips * mult * xMult);
  }

  identifyHandType(hand: Card[]): string {
    // Implementation of poker hand detection
    // (Check for Flush Five, Five of a Kind, Straight Flush, etc.)
    // Returns highest-ranking hand type found
  }

  getBaseChips(handType: string, level: number): number {
    // Base chips increase linearly with level
    const baseValues = {
      'High Card': 5,
      'Pair': 10,
      'Two Pair': 20,
      'Three of a Kind': 30,
      'Straight': 30,
      'Flush': 35,
      'Full House': 40,
      'Four of a Kind': 60,
      'Straight Flush': 100,
      'Flush Five': 160,
      'Flush House': 140,
      'Five of a Kind': 120
    };

    const chipGain = {
      'High Card': 10,
      'Pair': 15,
      'Two Pair': 20,
      // ... etc
    };

    return baseValues[handType] + chipGain[handType] * (level - 1);
  }

  // ... more methods
}
```

C. Recommendation Engine (TypeScript)

typescript

```
class RecommendationEngine {
  recommendBestPlay(gameState: GameState): Recommendation {
    const scoringEngine = new ScoringEngine();
    const allCombinations = this.generateAllCombinations(gameState.hand);

    let bestPlay = null;
    let bestScore = 0;

    for (const combo of allCombinations) {
      const score = scoringEngine.calculateScore(combo, gameState);

      if (score > bestScore) {
        bestScore = score;
        bestPlay = combo;
      }
    }

    return {
      cards: bestPlay,
      handType: scoringEngine.identifyHandType(bestPlay),
      expectedScore: bestScore,
      meetsTarget: bestScore >= gameState.targetScore
    };
  }

  recommendShopPurchase(gameState: GameState): Recommendation {
    const currentBuild = this.classifyBuild(gameState);
    const rankedJokers = [];

    for (const joker of gameState.shopJokers) {
      const synergy = this.calculateSynergy(joker, gameState.jokers, currentBuild);
      rankedJokers.push({ joker, synergy });
    }

    rankedJokers.sort((a, b) => b.synergy - a.synergy);

    return {
      recommendation: rankedJokers[0].joker,
      reasoning: this.explainSynergy(rankedJokers[0].joker, gameState),
      synergyScore: rankedJokers[0].synergy
    };
  }
  classifyBuild(gameState: GameState): string {
    // Analyze current Jokers and hand levels to determine strategy
    // Returns: "Flush", "Straight", "Mult-stack", "XMult-scaling", etc.
  }

  calculateSynergy(newJoker: Joker, existingJokers: Joker[], build: string): number {
    // Query synergy database
    // Calculate compatibility score (0-10)
  }
}
```

---

9. Conclusion

The Balatro AI Assistant is a **technically feasible** project with moderate complexity. The primary challenges are:

1. Real-time computer vision performance
2. Comprehensive strategy knowledge curation
3. Cross-platform compatibility

However, Balatro's 2D pixel art aesthetic, turn-based gameplay, and single-player nature make it an ideal candidate for AI assistance compared to fast-paced competitive games.

Recommendation: Start with a 2-4 week Proof of Concept to validate core assumptions, then decide whether to proceed with full development.

Expected Outcome: A useful tool that helps players learn optimal Balatro strategies, improves win rates, and deepens understanding of the game's mechanics.

Document Version: 1.0
Last Updated: December 2024
Author: Senior Solutions Architect Analysis