# Architectural Blueprint and Implementation Strategy for "Binky Medieval Showdown": Advanced Combat and Progression Systems

## 1. Executive Overview and Strategic Architectural Vision

The development trajectory of "Binky Medieval Showdown" necessitates a paradigm shift from a functional prototype to a mechanically sophisticated Action-RPG. The current operational state, characterized by generic click-to-attack interactions and shared input models, provides a baseline connectivity but fails to deliver the tactile "game feel" and strategic diversity required for a commercially viable product.[1] The objective of this comprehensive technical implementation plan is to engineer distinct gameplay loops for the Knight and Ninja archetypes, creating a bifurcation in player experience that ranges from heavy, physics-driven crowd control to high-velocity, precision-based mobility. Simultaneously, the progression interface—the Skill Tree—will be reimagined from a static UI overlay into an immersive, diegetic spatial environment known as the "Book of Ascension".[1]

This document serves as the definitive technical specification for this transformation. It leverages the Phaser 3 framework's extensive API surface to build modular, performant, and scalable systems. The central architectural thesis rests on decoupling input acquisition from action execution, enforcing a strict Command Pattern that mediates between the player's peripherals and the game's state machine. Furthermore, we will standardize combat interactions by adopting the "Skill Object" pattern—an architectural construct extrapolated from the Ogre enemy's logic—to treat every ability, whether a sword swing or a magic projectile, as a discrete, managed entity with its own lifecycle, physics properties, and visual logic.[1]

The scope of this report covers the granular implementation details of the Input Manager, the refactoring of the Player Controller into a state-driven entity, the mathematical and physics-based execution of class-specific skills, and the graphical engineering required for the new Skill Tree Scene. Every recommendation is grounded in the specific capabilities of the Phaser 3 engine, referencing its Input Plugin, Arcade Physics system, Camera Manager, and WebGL rendering pipeline.[1]

---

## 2. Core Systems Refactoring: The Abstracted Input

# Layer

The primary inhibitor to mechanical depth in the current build is the rigid coupling of input events (e.g., pointerdown) to specific function calls (e.g., performAttack). This direct binding prevents the implementation of complex mechanics such as channeled abilities, input buffering, or context-sensitive actions. To support the divergent needs of the Knight (Tank/CC) and Ninja (Glass Cannon/Mobility), we must abstract the hardware interface into a semantic action layer.

## 2.1 The Input Manager Architecture

The InputManager will serve as the gateway for all player interactions. Unlike simple event listeners, this manager will poll the state of the hardware and map it to a set of abstract PlayerAction enums. This allows the game logic to ask "Did the player intend to Dash?" rather than "Is the Shift key pressed?", facilitating easier remapping and gamepad integration later.

### 2.1.1 Semantic Mapping with the Input Plugin

Phaser's Input Plugin provides robust methods for handling keyboard and pointer data. We will utilize scene.input.keyboard.addKeys to create a consolidated object representing the input state.[1] This method allows for the simultaneous binding of multiple key codes to readable properties, which the InputManager can then query each frame.

For the Knight and Ninja, the inputs must be categorized into Primary, Secondary, and Special actions. The InputManager will maintain a mapping configuration:

- **Primary Action:** Mapped to pointer.leftButtonDown().[1] This controls the basic attack.
- **Secondary Action:** Mapped to pointer.rightButtonDown() or SHIFT. This triggers the Shield Bash or Shadow Dash.
- **Special Action:** Mapped to Q or E. This triggers the Whirlwind or Shuriken Fan.

Crucially, the manager must distinguish between an input that is currently held down and one that was just pressed. Phaser's Phaser.Input.Keyboard.JustDown(key) method is essential here.[1] For the Ninja's "Shadow Dash," which is an instantaneous resource-consuming action, we rely on JustDown to ensure the dash triggers exactly once per key press. Conversely, for the Knight's "Whirlwind," which is a channeled state, we check the isDown property of the key object to maintain the spinning state as long as the player holds the button.[1]

### 2.1.2 Input Buffering Implementation

In high-intensity combat, players often press buttons slightly before their character has finished the previous animation. If the game checks input strictly on the frame the character becomes idle, these inputs are lost, making the game feel unresponsive. To mitigate this, the InputManager will implement an Input Buffer.

The buffer functions by storing the timestamp of the last valid press for each action type. When a key is pressed, rather than triggering an immediate callback, the system records actionTimestamps = scene.time.now.

During the Player Controller's update cycle, instead of checking if the key is down right now, it

checks if the key was pressed within a specific tolerance window (e.g., the last 150ms).
if (scene.time.now - actionTimestamps < BUFFER_WINDOW)
If this condition is true and the player is in a valid state (e.g., IDLE or RUN), the action executes, and the timestamp is consumed (deleted). This creates a "sticky" feel to the controls, ensuring that a Shield Bash queued up during the final frames of a sword swing executes seamlessly the moment the swing completes.

### 2.2 Finite State Machine (FSM) Integration

The inputs processed by the InputManager drive the Player's State Machine. The current Player.ts likely relies on boolean flags, which becomes unmanageable with complex skills. We will transition to a formal FSM where the Player exists in mutually exclusive states: IDLE, RUN, ATTACK_PRIMARY, CASTING_SECONDARY, CHANNELING_SPECIAL, and STUNNED.
The transition logic is handled in the preUpdate or update loop.

- **From IDLE/RUN:** Valid inputs transition the player to ATTACK or CASTING states.
- **From CASTING:** The player is locked. Inputs are ignored or buffered. The exit condition is the completion of the animation or a specific timer event.
- **From CHANNELING:** Used for the Knight's Whirlwind. The state persists as long as the input is held (InputManager.isActionHeld). Movement logic is altered here; the standard velocity calculation is applied but multiplied by a damping factor (0.5), reflecting the "heavy" movement penalty of the skill.[1]

### 2.3 Cooldown and Resource Management

To enforce the tactical diversity of the classes, we cannot allow ability spamming. A CooldownManager will operate alongside the Input Manager. This system uses Phaser.Time.Clock to track availability.[1]
When an ability is successfully executed:

1. The CooldownManager records the nextReadyTime = scene.time.now + cooldownDuration.
2. It emits a COOLDOWN_START event via the Scene's Event Emitter (scene.events.emit).[1] The UI listens for this to start the overlay animation on the skill icon.

When the input is queried, the Player Controller first checks CooldownManager.isReady(Action.SECONDARY). If false, the input is ignored (or triggers a "not ready" sound cue). This separation of concerns ensures that the input logic remains pure, while the game rules regarding frequency are encapsulated in the manager.

---

# 3. The "Skill Object" Pattern: A Unified Combat Architecture

The existing implementation of the Ogre enemy uses a pattern where attacks are not merely animations played on the parent sprite, but separate logical entities (MeleeAttack.ts).[1] This is

the correct approach and must be standardized for all combat entities. We define this as the **Skill Object Pattern**.

## 3.1 Lifecycle of a Skill Object

A Skill Object is a Phaser.GameObjects.Container or Zone that encapsulates the entire logic of a single attack instance.[1] It decouples the visual representation of the attack (the slash effect, the projectile) from the actor casting it.
The lifecycle consists of five distinct phases:

1. **Instantiation:** The SkillSystem factory creates the object. It receives data: the caster (source of damage), the target stats (damage, knockback), and vectors (direction).
2. **Anticipation (Wind-up):** The object is active but the hitbox is disabled. This syncs with the "draw back" frames of the character's animation.
3. **Active (Impact):** Triggered by a timer or animation event frame. The Arcade Physics body is enabled. Visual Effects (VFX) like "swish" lines or particle bursts are played.
4. **Recovery:** The hitbox is disabled to prevent multiple hits from a single swing. Residual VFX fade out.
5. **Termination:** The object is destroyed or returned to an Object Pool.

## 3.2 Synchronization with Animation

Synchronizing the hitbox activation with the visual animation is critical for "game feel." Phaser's Animation system emits events that we can leverage. Specifically, we can add custom data to frames in the Texture Atlas or manually define event triggers.
However, a more robust code-driven approach, mirroring the Ogre implementation [1], uses scene.time.delayedCall. When the Knight initiates ShieldBash, the animation Knight_1_AttackRun begins. Simultaneously, the ShieldBash Skill Object is created. This object knows that the "impact" frame of the animation occurs 400ms into the sequence. It sets an internal timer. When the timer fires, the object calls this.body.setEnable(true) [1], activating the hitbox exactly when the shield is fully extended visually. This deterministic approach avoids the fragility of frame-based event listeners which can sometimes be skipped if the frame rate dips.

## 3.3 Collision Handling via Groups

To manage the performance of potentially dozens of active skills (shurikens, slashes, explosions), we utilize Phaser.GameObjects.Group.[1] The PlayerSkillSystem maintains a Group for all active player hitboxes.
In the Scene's update loop, collision interaction is defined globally:

JavaScript

```
this.physics.add.overlap(
  this.skillSystem.hitboxGroup,
```

```
    this.enemyGroup,
    this.handleSkillHit,
    this.processSkillHit,
    this
);
```

This single overlap call leverages Arcade Physics' spatial partitioning (QuadTree) to efficiently check all active player attacks against all enemies. The processSkillHit callback can be used for additional logic, such as checking if an enemy is currently invulnerable or if the projectile has already hit its max target count (piercing logic).

---

# 4. The Knight Archetype: Implementing Weight and Crowd Control

The Knight design philosophy centers on "Weight," "Commitment," and "Crowd Control".[1] The technical implementation must reinforce these pillars through physics manipulation and camera effects.

## 4.1 Secondary Ability: Shield Bash

Design Goal: A heavy, directional slam that stuns enemies.
Visuals: Uses Knight_1_AttackRun animation; spawns bone-slam VFX.1

### 4.1.1 Conical Hitbox Detection

Arcade Physics supports Axis-Aligned Bounding Boxes (AABB) and Circles. It does not natively support Cones. To implement the Shield Bash's area of effect, we must employ a hybrid approach using a Circular Body and vector math.

1. **The Range Check:** The Skill Object spawns a circular physics body centered on the Knight. The radius represents the reach of the bash.
2. **The Angle Check:** When the physics engine reports an overlap between an enemy and this circle, we perform a secondary verification in the callback.
    - We calculate the vector from the Knight to the Enemy: toEnemy = enemy.position - knight.position.
    - We normalize this vector.
    - We take the dot product of toEnemy and the Knight's facingVector.
    - If the dot product is greater than a threshold (e.g., 0.707 for a 45-degree half-angle), the enemy is within the "Cone" in front of the Knight.

This technique allows us to maintain the efficiency of Arcade Physics for the broad-phase check while simulating complex geometry for the gameplay logic.

### 4.1.2 The Stun Mechanic

Upon a validated hit, the system applies a "Stun" state to the enemy. This is a data-driven

interaction.

1. **Flagging:** The enemy instance receives enemy.setStunned(duration).
2. **Physics Halt:** enemy.body.setVelocity(0, 0) is called immediately to stop their movement.[1]
3. **State Lock:** The Enemy AI's update loop checks if (this.isStunned) return;, effectively skipping all logic for pathfinding or attacking.
4. **Visual Feedback:** A "dizzy" particle effect or icon is spawned above the enemy using a separate Container.

### 4.1.3 Impact Feedback (Juice)

To sell the "Weight" of the attack, we use the Camera Manager. On the frame the hitbox activates:
this.cameras.main.shake(100, 0.01).1
This slight vibration communicates impact force. Additionally, the Knight's own velocity is set to zero during the cast time, forcing the player to commit to the move and reinforcing the "tank" archetype—he plants his feet to strike.

## 4.2 Ultimate Ability: Whirlwind

Design Goal: A continuous, mobile Area of Effect (AoE) attack.
Visuals: Spinning sprite, blurred trails.

### 4.2.1 State-Driven Physics

Unlike the Shield Bash, the Whirlwind allows movement. This requires a modification to the Player Controller's state machine.
When in the WHIRLWIND state:

1. **Input Scaling:** The input vector from the InputManager is not applied directly to velocity. Instead, it is scaled: body.setVelocity(input.x * speed * 0.5, input.y * speed * 0.5).[1] This 50% speed reduction is crucial for balance and feel.
2. **Animation Override:** The sprite's rotation property or animation is continuously updated to loop the spin visuals.

### 4.2.2 Persistent Hitbox Logic

The Whirlwind hitbox is a PlayerNova object 1 that must track with the player.
While Container objects can group sprites, putting a Physics Body inside a moving Container can sometimes yield synchronization issues in Arcade Physics depending on how the world transform is updated. A robust alternative is to have the PlayerNova be an independent Game Object in the scene that creates a "Hard Lock" in its preUpdate method: this.setPosition(this.player.x, this.player.y).
Damage Ticking:
Since the overlap check runs every frame (60Hz), applying damage on every overlap would instantly kill enemies. The SkillObject must implement a "Tick Rate."

- It maintains a Map<EnemyID, number> tracking the lastHitTime for each enemy in range.
- In the collision callback: if (now - lastHitTime > damageInterval) { applyDamage();

lastHitTime = now; }.
This ensures damage applies rhythmically (e.g., every 250ms) regardless of frame rate.

---

# 5. The Ninja Archetype: Speed and Object Pooling

The Ninja archetype demands "Precision," "Burst," and "Mobility".[1] The technical challenge here shifts from state management to high-frequency object manipulation and tweening systems.

## 5.1 Secondary Ability: Shadow Dash

Design Goal: Instant linear movement that damages enemies passed through.
Visuals: Fade effect, afterimages.

### 5.1.1 Physics vs. Tweening

Using standard velocity (body.setVelocity(highValue)) for a dash acts poorly in Arcade Physics. If the speed is high enough, the object might "tunnel" through thin walls between frames. Furthermore, friction and drag calculations can make the distance inconsistent. Therefore, we use Phaser Tweens for the movement.1 Tweens provide deterministic start and end points over a fixed duration.

### 5.1.2 Raycasting for Safety

Before tweening, we must ensure the Ninja doesn't dash inside a wall.

1. **Ray Calculation:** We create a Phaser.Geom.Line from the Ninja's center to the target cursor position.
2. **Collision Check:** We iterate along this line (or use physics.world.raycast if available in the specific plugin version, otherwise we sample points along the line). We check these points against the collision layer of the Tilemap.
3. **Clamping:** If a wall is found at distance D, the dash target is clamped to D - playerRadius. This ensures the dash always ends in valid space.

### 5.1.3 The "Beam" Hitbox

Since the Ninja moves rapidly (teleports, effectively), a standard body carried with him might miss enemies if the tween updates position in large jumps.
Instead of a moving hitbox, we spawn a Static Linear Hitbox.

- We create a rectangular physics body that spans the entire length of the dash vector.
- Width = Dash Distance. Height = Character Width.
- Rotation = Angle of Dash.
- This hitbox exists for a brief window (e.g., 100ms) to register hits on everything in the path, then dissipates.

### 5.1.4 Invulnerability and Ghosting

During the Tween's onStart callback, we set player.isInvulnerable = true. In onComplete, we

reset it.
To create the "Shadow" visual:

- We create a TimerEvent that repeats every 50ms during the dash.
- The callback spawns a static Image of the Ninja at his current location.
- This image is tinted black/purple (0x550055) [1] and has its alpha tweened from 0.5 to 0 over 300ms.
- This creates a trail of fading silhouettes, visually explaining the "Shadow Dash" name.

## 5.2 Primary Variant: Shuriken Fan

Design Goal: A spread of projectiles.
Visuals: 3 rotating shurikens.

### 5.2.1 Object Pooling Strategy

The Ninja attacks rapidly. Instantiating new Sprite objects for every shuriken will lead to memory fragmentation and Garbage Collection spikes, causing frame drops. We must use Object Pooling.
We use a Phaser.GameObjects.Group configured for pooling 1:

JavaScript

```
this.shurikens = this.physics.add.group({
  classType: Shuriken,
  maxSize: 50,
  runChildUpdate: true
});
```

When the player fires:

1. Call shurikens.get(x, y). This searches for an inactive (dead) shuriken in the pool. If none exist and maxSize isn't reached, it creates one.
2. Call shuriken.fire(angle). This method resets the texture, enables the body, and sets the velocity.

### 5.2.2 The Fan Algorithm

To create the fan spread:

1. Calculate the baseAngle to the cursor using Phaser.Math.Angle.Between.
2. Define the spread (e.g., 15 degrees or ~0.26 radians).
3. Launch three projectiles with velocity vectors derived from: baseAngle - spread, baseAngle, and baseAngle + spread.
4. Apply setAngularVelocity(300) to the sprites to make them spin visually as they travel.[1]

---

# 6. Visual Effects and Feedback Systems ("Juice")

"Juice" refers to the non-functional feedback that makes interactions feel responsive. We will leverage Phaser's FX components and Particle Emitters.

## 6.1 Particle Systems

Phaser 3.60+ introduced a revamped ParticleEmitter.[1] We will use this for hit impacts and dashes.

- **Impact Flares:** When a Shuriken hits an enemy, we emit a "burst" of 5-10 particles. We use the explode() method on the emitter rather than continuous flow.
- **Blood/Dust:** For the Knight's slam, a radial emitter placed at the impact point sends "dust" particles outward, using a high drag value so they slow down quickly, simulating debris.

## 6.2 Post-Processing (FX)

We will utilize the PostFX pipeline for the Ninja.[1]

- **Bloom:** When the Ninja charges a special attack, we can apply a Bloom effect (gameObject.postFX.addBloom()) to the character sprite, causing them to glow. This provides immediate visual readability of the "powered up" state.
- **ColorMatrix:** When the game is paused or the Skill Tree is open, we can apply a ColorMatrix filter to the GameScene camera to desaturate it (grayscale()), visually receding the game world to the background.

## 6.3 Camera Manipulation

Beyond simple shakes, we will implement Camera Lag.
Instead of strictly locking the camera to the player (startFollow), we use lerp (Linear Interpolation) values like 0.1 on the follow style.1 This makes the camera trail slightly behind the player, smoothing out jittery movement.
During high-speed moves like the Shadow Dash, we can momentarily increase the lerp value or apply a zoom effect (zoom in slightly on impact) to accentuate the action.

---

# 7. Skill Tree UI Architecture: The "Book of Ascension"

The requirement is to move from a static menu to an immersive, explorable space.[1] This dictates that the Skill Tree cannot simply be a UI overlay; it must be a fully realized **Scene**.

## 7.1 Parallel Scene Architecture

Phaser supports multiple active scenes. The SkillTreeScene will run in parallel to the GameScene.

- **Toggle Logic:** When 'T' is pressed, the GameScene is *not* stopped. Instead, we call this.scene.pause('GameScene') or simply this.scene.setVisible(false, 'GameScene') if we

want to hide it completely. Alternatively, we keep it visible but apply the blur/grayscale shader mentioned above, rendering the Skill Tree on top.

- **Input Isolation:** The InputManager must know to stop processing combat inputs. Launching the Skill Tree Scene should trigger a scene.input.stopPropagation() or the GameScene should check scene.isActive() before processing clicks.

## 7.2 The Infinite Canvas: Camera Implementation

The Skill Tree will simulate a large, scrollable map (constellations or a parchment). We do not move the UI elements; we move the Camera.

- Navigation: We bind pointer drag events to the Camera's scrollX and scrollY.
  camera.scrollX -= (pointer.x - pointer.prevPosition.x) / camera.zoom.1
  The division by zoom is crucial; otherwise, panning feels too fast when zoomed in.
- **Zoom:** We bind the wheel event to camera.setZoom(). We clamp the values between 0.5 (overview) and 2.0 (detail).

## 7.3 Parallax Backgrounds

To create depth, the background of the Skill Tree (e.g., a nebula or table texture) uses a lower scroll factor.
this.add.image(0, 0, 'nebula').setScrollFactor(0.5).1
This means for every 100 pixels the camera moves, the background only moves 50, creating a pseudo-3D depth effect that makes the skill nodes feel like they are floating above the surface.

## 7.4 Dynamic Connection Lines

Drawing connections between hundreds of nodes using Sprite images is inefficient. We use Phaser.GameObjects.Graphics.[1]

- We create a single Graphics object for the entire tree.
- We iterate through the node data. graphics.lineBetween(nodeA.x, nodeA.y, nodeB.x, nodeB.y).
- **State Styling:** We check the unlock status of the connection.
  - **Locked:** Thin, grey line, 0.3 alpha.
  - **Unlockable:** Thicker, white line, pulsing alpha (using a Tween).
  - Unlocked: Gold line, full opacity.
    This graphics object is redrawn only when a skill is purchased, minimizing overhead.

## 7.5 DOM Elements for Rich UI

For tooltips, Phaser's built-in text rendering (BitmapText) handles word wrapping and formatting poorly compared to HTML/CSS.
We will use this.add.dom(x, y).createFromCache('tooltip_template').1

- When a node is hovered, we instantiate this DOM element.
- We populate its HTML content dynamically: element.getChildByID('title').innerText =

skill.name.
- This allows us to use CSS for complex layouts, borders, shadows, and even embedding small GIFs/Videos of the skill in action within the tooltip itself.

---

# 8. Data Persistence and Schema

The depth of the system requires a structured approach to data. Hardcoded values in Player.ts [1] must be replaced with external data definitions.

## 8.1 Data Schema

We will define the Skill Tree in a JSON format loaded via this.load.json.[1]

JSON

```
{
  "skills":,
     "cost": 100,
     "effect": { "unlockAbility": "SHIELD_BASH", "damageMult": 1.5 }
   }
 ]
}
```

## 8.2 The Registry and LocalStorage

Phaser's Registry (this.registry) is a global data store perfect for this.[1]

- **Runtime:** The game loads the JSON into the registry. The PlayerProfile object in the registry tracks unlockedSkillIDs.
- **Events:** When a skill is bought, we emit registry.events.emit('unlock_skill', id). The Player Controller listens for this to dynamically add the new Action to the InputManager mapping.
- **Persistence:** On save (or auto-save), we serialize the PlayerProfile from the Registry to window.localStorage.

---

# 9. Performance Optimization Strategy

Implementing these features adds significant load. We must proactively manage performance.

## 9.1 Texture Packing

All new assets—Skill Icons, Particle Flares, Character Frames—must be packed into **Texture**

**Atlases**.[1] This allows Phaser to batch the rendering calls. Instead of 50 draw calls for 50 skill icons, an Atlas allows them to be drawn in 1 call if they share the same texture source.

### 9.2 Render Texture Caching

For the Skill Tree, if the background and static lines are complex, we can render them once to a RenderTexture [1] and simply display that single texture, rather than re-calculating the graphics paths every frame. However, given the need for dynamic "pulsing" lines, the Graphics approach is preferred unless node counts exceed 500+.

### 9.3 Event Cleanup

The InputManager and SkillObjects rely heavily on Event Emitters. It is imperative that destroy() methods are implemented robustly.

- this.input.off() must be called when the Scene shuts down.
- Skill Objects returning to the pool must call this.off() on any timers or physics collisions they established. Failure to do so will cause "ghost" interactions where invisible, destroyed hitboxes still trigger damage logic.

---

## 10. Implementation Roadmap

### Phase 1: Foundation (Input & State)

1. Refactor Player.ts to implement the Finite State Machine.
2. Build the InputManager class with Action Mapping and Buffering.
3. Establish the SkillObject base class and the PlayerSkillSystem factory.

### Phase 2: Mechanics (The "Feel")

1. Implement Knight's ShieldBash using the conical check and camera shake.
2. Implement Ninja's ShadowDash using Tweens and raycasting.
3. Implement Ninja's ShurikenFan using Group Pooling.

### Phase 3: The Interface (Skill Tree)

1. Create SkillTreeScene with parallel execution logic.
2. Implement Camera Panning/Zooming and Parallax background.
3. Implement Procedural Line Drawing and DOM-based Tooltips.

### Phase 4: Integration

1. Connect the Registry data to the Player Controller (skills only work if unlocked).
2. Final polish pass on VFX (Particles, PostFX Bloom).

By strictly adhering to this architectural blueprint, the engineering team will avoid common pitfalls such as input lag, physics tunneling, and spaghetti code state management. The result will be a robust, extensible codebase capable of supporting the "Juice" and "Depth"

envisioned for "Binky Medieval Showdown."

## Works cited

1. GEMINI_DEEP_RESEARCH_PLAN.txt