Morlocs Tower Defense - Technical Design Document

1. Core Functionality

A tower defense game where creeps spawn from fixed points on the battlefield and move in a straight line toward a central base. The player places turrets on the battlefield to shoot and destroy creeps before they reach the base. Creeps arrive in waves of increasing difficulty. The player earns coins from killing creeps and spends coins to place turrets. The game is won by surviving all waves; the game is lost when the base takes too much damage.

Key Systems

- **Creep Spawning & Movement** - Creeps spawn from pre-placed SpawnPoints and move in a straight line toward the Base (no pathfinding). Spawn timing, count, and behavior are data-driven and easy to tune.
- **Base Health & Lose Condition** - The Base has a health pool. Creeps that reach it deal damage. When health reaches zero, the LosePopup is displayed.
- **Turret Placement** - The player clicks on the battlefield to place turrets. Placement costs coins from the economy system.
- **Turret Targeting & Projectiles** - Turrets detect creeps within range and fire projectiles that deal damage on hit. Two turret types: regular (damage) and freezing (slow effect).
- **Creep Variety** - Two creep types (small, big) with differing speed and hit points.
- **Economy** - Turrets cost coins to build. Creeps award coins on death. Starting coins provided.
- **Wave System** - Creeps arrive in defined waves. Clearing all creeps in a wave triggers the next. Surviving all waves displays the WinPopup.
- **Game Reset** - Full restart without leaving Play Mode. All state (enemies, towers, resources, wave progress) resets cleanly.

---

2. Architecture

Guiding Principles

These are architectural directions that will inform the detailed design. Not full ECS (Unity DOTS is out of scope for this project and Unity version), but borrowing key ideas:

- **Data-oriented design** - Separate data from behavior. Game state lives in plain data structures (structs, ScriptableObjects); systems operate on that data. Prefer structs for hot-path data. Minimize scattered state across MonoBehaviours.
- **System-and-component thinking** - MonoBehaviours act as thin components that hold references and wire into Unity lifecycle. Logic lives in systems/managers that process components, not in the components themselves.
- **MonoBehaviour minimalism** - Only inherit from MonoBehaviour when Unity requires it (scene presence, coroutines, serialized inspector references, collision callbacks). Pure logic, data models, state machines, and utility classes should be plain C# classes or structs. Avoid an architecture dominated by per-object `Update()` calls; prefer centralized system ticks that iterate over data.
- **Simulation/presentation separation** - The game simulation (state, logic, rules) is independent of visuals. Simulation ticks on data and produces state changes; the presentation layer reads state and updates transforms, effects, and UI. No gameplay logic should depend on visual state. Architect as if a headless server could run the sim -- this also enables Edit Mode testing of game logic without a scene.
- **MVU-style UI architecture** - UI follows a Model-View-Update pattern: a model (state) drives what the view displays; user actions produce messages/events that update the model; the view re-renders from the model. No direct UI-to-game-state mutation.
- **State machine state management** - Game flow (start, playing, wave transitions, win/lose, reset) managed by an explicit state machine. Lightweight custom implementation -- no Stateless library (avoids LINQ and allocation concerns). States are data, transitions are explicit.

Detailed Design

See Architecture Diagrams for class diagrams, state diagrams, and sequence diagrams.

Bootstrap & Game Loop

A single `GameBootstrap` MonoBehaviour in MainScene serves as the entry point and composition root. It creates the `GameStateMachine`, all `IGameState` implementations, and all gameplay systems. `Update()` delegates to the state machine, which ticks the current state.

```
GameBootstrap.Update() → GameStateMachine.Tick() → IGameState.Tick() → [System ticks in PlayingState]
```

Class Responsibilities

Class	Type	Responsibility
`GameBootstrap`	MonoBehaviour	Composition root. Creates state machine, states, systems. Configures transition table. Drives game loop via `Update()`.
`GameStateMachine`	Plain C#	Owns state registry and transition table. Resolves triggers to transitions. Fires `OnStateChanged` event. Processes pending triggers at tick start.
`GameState`	Enum	State identifiers: `Init`, `Playing`, `Win`, `Lose`.
`GameTrigger`	Enum	Semantic transition triggers: `SceneValidated`, `BaseDestroyed`, `AllWavesCleared`, `RestartRequested`.
`IGameState`	Interface	Contract for game states: `Enter()`, `Tick(float)`, `Exit()`.
`InitState`	Plain C# : `IGameState`	Validates scene setup (Base, SpawnPoints). Fires `SceneValidated`. Future: async Addressable loading.
`PlayingState`	Plain C# : `IGameState`	Ticks `IGameSystem[]` in registered order. Fires `BaseDestroyed` / `AllWavesCleared` when conditions met.
`IGameSystem`	Interface	Contract for gameplay systems: `Tick(float)`.
`BaseComponent`	MonoBehaviour	Thin component on Base GameObject. Identifies the base for system discovery. Future stories add health data.

Game State Machine

Trigger-based transitions — inspired by sn-unity FLUX's explicit state machine pattern. States fire semantic triggers describing *what happened*; the state machine resolves triggers against a declarative transition table to determine *where to go*. States never reference other states directly.

The transition table is configured in `GameBootstrap` — the single place where the full game flow is visible:

```
AddTransition(Init,    SceneValidated,  Playing)
AddTransition(Playing, BaseDestroyed,   Lose)
AddTransition(Playing, AllWavesCleared, Win)
AddTransition(Win,     RestartRequested, Init)
AddTransition(Lose,    RestartRequested, Init)
```

States receive an `Action<GameTrigger>` delegate at construction. Calling `fire(GameTrigger.SceneValidated)` sets a pending trigger. The state machine resolves pending triggers at the start of `Tick()`: lookup `(currentState, trigger)` → if valid transition exists → `Exit()` current → switch → `Enter()` new → `Tick()` new. Invalid triggers are logged and ignored. This guarantees a clean frame boundary between states.

Single-pending-trigger constraint: `GameStateMachine` stores at most one pending trigger (`GameTrigger?`). If a state fires a second trigger before the first is resolved, the second overwrites the first and a warning is logged. This is intentional — end conditions are evaluated after all systems have settled within a single tick, and the conditions are mutually exclusive (the base cannot be both destroyed and alive when all waves clear). The single-trigger design keeps the state machine simple and avoids

queue-ordering complexity. If a future project needed multiple simultaneous triggers, the field could be replaced with a queue — but for this game's state graph, it is unnecessary.

States: `Init` → `Playing` → `Win` / `Lose` → `Init` (restart)

Reset: Transitioning back to `Init` triggers `PlayingState.Exit()` (tear down spawned objects, reset systems), then `InitState.Enter()` (re-validate scene, set up fresh game).

Component & System Pattern

- **Scene components** (MonoBehaviours): Thin scene presence. Hold serialized references, identify GameObjects for system discovery, no game logic. Examples: `BaseComponent`, `CreepComponent`, `TurretComponent`, `SpawnPointComponent`. Throughout this document, "component" always refers to these MonoBehaviour scene markers — not simulation data.
- **Simulation data** (plain C# classes, structs, or arrays): The authoritative game state that systems read and write. Positions, health, targets, economy balance, status effects. No Unity types (`Transform`, `GameObject`) in sim data. At this project's entity counts (tens of creeps, tens of turrets), classes are the pragmatic default — they avoid the copy-modify-write-back friction of mutable structs stored in collections. Reserve structs for small, immutable value types (e.g., `GridPosition`) or cases where cache locality measurably matters.
- **Systems** (plain C# classes implementing `IGameSystem`): Own all game logic and the simulation data for their domain. Created by `GameBootstrap`, registered with `PlayingState` as an ordered array, ticked in deterministic order. Examples: `SpawnSystem`, `MovementSystem`, `TargetingSystem`, `DamageSystem`, `EconomySystem`, `WaveSystem`.

The system tick order is configured in `GameBootstrap` — the composition root is the single source of truth for both the transition table and the system execution order:

```
new PlayingState(stateMachine.Fire, new IGameSystem[]
{
    // Phase 1 — World Update
    waveSystem, spawnSystem, movementSystem, placementSystem,
    // Phase 2 — Combat
    targetingSystem, projectileSystem, damageSystem,
    // Phase 3 — Resolution
    economySystem
})
```

Systems operate on plain simulation data (structs, arrays) — not on MonoBehaviour references directly. A thin presentation adapter layer syncs simulation state to/from scene objects each frame. Components do not contain game logic.

Event Messaging

- C# `event Action<T>` for system-to-system and system-to-UI communication
- `?.Invoke()` for safe invocation
- Events are notifications, not commands — receivers do not mutate the sender's state
- Key events: `OnStateChanged`, `OnCreepKilled`, `OnCoinsChanged`, `OnBaseHealthChanged`, `OnWaveStarted`, `OnWaveCleared`

Ordering & reentrancy policy:

- Events fire inline during the producing system's `Tick()`. Because systems run sequentially in a deterministic order, event delivery order is deterministic. Within a single event invocation, C# multicast delegates (`event Action<T>`) invoke subscribers in subscription order (FIFO). Since all subscriptions are established at bootstrap time in a fixed order, listener execution order is also deterministic.
- **Handlers must not fire game triggers or mutate simulation state they don't own.** A handler may buffer data locally (e.g., EconomySystem records a pending coin credit when it hears `OnCreepKilled`) and applies it during its own `Tick()`.
- UI listens to events for display updates only — never writes back to simulation state. Player actions flow through an input struct read by the relevant system (e.g., `PlacementSystem`).

- If future complexity demands decoupling, events can be replaced with a per-frame message queue. For this project's scale, inline `Action<T>` with the handler discipline above is sufficient.

Simulation / Presentation Boundary

The simulation and presentation are separated by an explicit data boundary:

- **Simulation model** — Plain C# data (structs, arrays) owned by systems. All gameplay state lives here: creep positions, health values, turret targets, projectile positions, economy balance. Systems read and write *only* this data. No `Transform`, no `GameObject`, no MonoBehaviour references in system code.
- **Presentation adapter** — A thin layer that runs after all system ticks. Reads simulation state and writes to Unity objects (`Transform.position`, enable/disable GameObjects, UI updates). Also reads Unity inputs that the sim needs (e.g., mouse position via raycast) and writes them into sim-readable input structs *before* systems tick.
- **Boundary rule** — Data flows one direction per phase: Unity → input structs → systems → sim state → presentation adapter → Unity. Systems never call Unity APIs directly.

Testability implication: Edit Mode tests create simulation data, tick systems, and assert on results — no GameObjects, no scene, no Play Mode required. Any system that needs information from Unity (physics queries, input) receives it through an interface that tests can stub.

System Data Ownership

Each simulation data domain has a single authoritative system — the only system allowed to write that data. Other systems may read it but never mutate it directly. This is the tower defense equivalent of the "single-writer store" principle from the FLUX analysis, and it prevents hidden cross-mutation and ordering-dependent bugs.

Data Domain	Owner (Writer)	Readers
Creep spawn queue	WaveSystem	SpawnSystem
Active creep registry	SpawnSystem	MovementSystem, TargetingSystem, DamageSystem
Creep positions & velocity	MovementSystem	TargetingSystem, DamageSystem, PresentationAdapter
Creep health	DamageSystem	EconomySystem (death event), PresentationAdapter
Turret placement & positions	PlacementSystem	TargetingSystem, PresentationAdapter
Turret target assignments	TargetingSystem	ProjectileSystem
Projectile positions & state	ProjectileSystem	DamageSystem, PresentationAdapter
Base health	DamageSystem	PlayingState (end condition), PresentationAdapter
Coin balance	EconomySystem	PlacementSystem (affordability), PresentationAdapter
Wave progress	WaveSystem	PlayingState (end condition), PresentationAdapter
Player input (placement)	PresentationAdapter	PlacementSystem
Active status effects (slow, etc.)	DamageSystem	MovementSystem

Cross-system communication happens through the deterministic tick order, not through event-driven mutation. Example: when DamageSystem kills a creep, it fires `OnCreepKilled`. EconomySystem hears this event and buffers a pending coin credit — but it does not write to creep data or economy state during the handler. It applies the credit during its own `Tick()`. This guarantees that system execution order, not subscription order, determines when state changes take effect.

Status effect pattern (Story 7): The freezing turret introduces a cross-system concern — DamageSystem applies a slow effect, but MovementSystem owns creep velocity. To preserve single-writer discipline: DamageSystem writes status effect records (effect type, duration, remaining time) to a per-creep status effect list it owns. MovementSystem reads those records during its tick and adjusts speed accordingly. DamageSystem never writes to velocity; MovementSystem never writes to status effects. The phase ordering guarantees that Combat (Phase 2) writes effects before the next frame's World Update (Phase 1) reads them.

MVU UI Boundaries

- **Model**: Simulation state exposed via read-only properties (coins, base health, wave number, selected turret type)
- **View**: UI Toolkit documents bound to model data; provided UGUI popups toggled by game state events
- **Update**: Player actions → commands/events → systems process → model changes → view re-reads

No direct UI-to-game-state mutation.

Object Pooling (Story 2+)

Creeps and projectiles use pre-allocated object pools managed by their respective systems. `IPoolable` interface provides consistent `OnPoolGet()` / `OnPoolReturn()` lifecycle. Pools `Clear()` and rebuild on game reset — no residual allocations.

Folder Organization

```
Assets/Scripts/
├── Core/               # GameBootstrap, GameStateMachine, GameState, GameTrigger, IGameState,
IGameSystem, states
├── Base/               # BaseComponent
├── Creeps/             # (Story 2+) SpawnPointComponent, SpawnSystem, MovementSystem
├── Turrets/            # (Story 4+) TurretComponent, PlacementSystem, TargetingSystem
├── Combat/             # (Story 5+) ProjectileSystem, DamageSystem
├── Economy/            # (Story 6+) EconomySystem
├── Waves/              # (Story 9) WaveSystem
├── Data/               # ScriptableObject definitions (CreepDef, TurretDef, WaveDef, EconomyConfig)
└── UI/                 # UI Toolkit binding, HUD controller
```

No project-wide namespace. Feature folders group related components, systems, and data.

Extension Points

- **New creep/turret variant** (same behavior, different stats): Add ScriptableObject definition asset → Addressables picks it up → existing systems process it. Data-only change.
- **New creep/turret behavior** (e.g., a novel turret effect): Requires a new effect handler in the relevant system(s). Scope depends on how different the behavior is from existing types.
- **New game state**: Implement `IGameState` → add to `GameState` enum → register with `GameStateMachine` → add transition rows in `GameBootstrap`.
- **New trigger**: Add to `GameTrigger` enum → add transition rows in `GameBootstrap`. Existing states unchanged.
- **New gameplay system**: Implement `IGameSystem` → add to `PlayingState` system array in `GameBootstrap` at the correct tick position.

---

3. Constraints & Ground Rules

These constraints come directly from the spec and must be respected during implementation.

- **No pathfinding** - Creeps move in a straight line toward the Base. Do not implement or integrate NavMesh or any pathfinding system.
- **Editor-only** - The game will be tested in the Unity Editor. No platform build or deployment concerns.
- **Mouse & keyboard input** - Must support mouse and keyboard controls.
- **No extra features** - Only implement the 9 requirements listed in the spec. No bonus features.
- **No visual polish** - Code quality and architecture are evaluated, not visual fidelity. Don't invest in aesthetics.
- **Scalability focus** - Architecture must make it easy to add new unit or turret types without major refactors.
- **Use provided assets** - Use the prefabs and materials already in the project. Do not create replacement assets.
- **No Unity asset creation from code tools** - Claude Code must not create Unity assets (scenes, prefabs, ScriptableObjects, materials, etc.). These are created in the Unity Editor by the developer. Claude Code writes C# scripts, edits existing scripts, and creates non-asset files only. When a ScriptableObject class is written in code, the developer will create the corresponding `.asset` file in the Editor.

- **Unity version** - 2022.3.58f1.

---

4. Tech Package Choices

Package	Purpose	Notes
Input System (new)	Player input (mouse clicks, keyboard)	Use `UnityEngine.InputSystem`. Do not use legacy `Input` class.
UI Toolkit	New runtime UI (HUD, overlays)	UXML for layout, USS for styles, C# for binding. Only for UI not already provided as prefabs.
UGUI	Provided popup prefabs	WinPopup and LosePopup are provided as UGUI prefabs. Use as-is; do not rebuild in UI Toolkit.
ScriptableObjects	Tuning data	Creep definitions, turret definitions, wave definitions, economy config.
TextMeshPro	*Not used*	Provided UGUI popups use legacy Text. New UI uses UI Toolkit labels.
Cinemachine	*Not used*	No camera control needed for this project.
Addressables	ScriptableObject data loading	Load tuning data (creep defs, turret defs, wave defs, economy config) via Addressables for extensibility. Visual prefabs use direct serialized field references.

Asset Loading Strategy

- ScriptableObject tuning data loaded via Addressables -- enables adding new definitions without code changes
- Visual prefabs (creeps, turrets, base, popups) referenced directly via serialized fields
- Avoid `Resources/` folder; use Addressables or direct references

---

5. Data Configuration Strategy

The spec emphasizes making gameplay values "easy to tweak and tune." This section defines how tunable data is exposed.

To be filled out during architecture discussion.

---

6. Provided Assets Reference

Assets already present in the Unity project:

Asset	Type	Location
MainScene	Scene	Assets/Scenes/
Base	Prefab	Assets/Prefabs/
SpawnPoint - 1	Prefab	Assets/Prefabs/
Turret-regular	Prefab	Assets/Prefabs/
Turret-freezing	Prefab	Assets/Prefabs/
Creep-small	Prefab	Assets/Prefabs/
Creep-big	Prefab	Assets/Prefabs/
WinPopup	Prefab	Assets/Prefabs/UI/

Asset	Type	Location
LosePopup	Prefab	Assets/Prefabs/UI/
Terrain	Asset	Assets/Terrain/
Various materials	Materials	Assets/Materials/

The MainScene already contains: Terrain, Base (centered), and SpawnPoints placed on the battlefield.

---

7. Deliverables - User Stories

Implementation order is designed to build systems incrementally, with each story producing a testable result.

Story 1: Project Foundation

> *As a developer, the project has a working architectural skeleton so gameplay systems can be built on a solid base.*

Acceptance Criteria:

- Game bootstrap / entry point exists and runs on Play
- Game state machine is implemented with at least `Init` and `Playing` states
- Base has a component that systems can discover
- Folder structure is established under `Assets/Scripts/`
- Test infrastructure is set up (Edit Mode and Play Mode test assemblies exist and run)
- A test verifies the game state machine transitions from `Init` to `Playing`

Story 2: Creep Spawning & Movement

> *As a player, I see creeps spawn from the edges of the battlefield and move toward my base.*

Acceptance Criteria:

- Creeps instantiate from the pre-placed SpawnPoint positions at configurable intervals
- Creeps move in a straight line toward the Base position (no pathfinding)
- Spawn timing, creep count per spawn, and movement speed are exposed as tunable parameters
- Creeps use the provided Creep-small prefab

Story 3: Base Health & Lose Condition

> *As a player, when too many creeps reach my base, I am informed I have lost.*

Acceptance Criteria:

- The Base has a configurable health value
- When a creep reaches the Base, it deals damage to the Base and is destroyed
- When Base health reaches zero, the LosePopup prefab is displayed
- A health bar or health indicator is visible for the Base (or creeps -- spec allows either)

Story 4: Turret Placement

> *As a player, I can place turrets on the battlefield using my mouse.*

Acceptance Criteria:

- Clicking on the battlefield instantiates a turret at the clicked position
- The turret uses the provided Turret-regular prefab
- Turrets remain stationary after placement

- Placement works via mouse input on the terrain

Story 5: Turret Shooting & Creep Damage

> *As a player, turrets I've placed automatically shoot at nearby creeps and destroy them.*

Acceptance Criteria:

- Turrets detect creeps within a configurable range
- Turrets fire projectiles at a configurable rate toward the nearest creep in range
- Projectiles travel toward the target and deal configurable damage on hit
- Creeps are destroyed when their HP reaches zero
- Damage amount and creep HP are exposed as tunable parameters

---

Story 6: Economy System

> *As a player, I spend coins to place turrets and earn coins from killing creeps.*

Acceptance Criteria:

- Player starts with a configurable amount of coins
- Each turret costs 5 coins to place (configurable)
- Each creep awards 1 coin on death (configurable)
- Player cannot place a turret if they lack sufficient coins
- Current coin count is displayed in the UI

Story 7: Turret Types (Regular & Freezing)

> *As a player, I can choose between a regular turret and a freezing turret, each with different effects.*

Acceptance Criteria:

- Two turret types are available: Regular and Freezing
- Regular turret deals direct damage (as already implemented)
- Freezing turret applies a slow effect to creeps it hits, reducing their movement speed for a configurable duration
- Player can select which turret type to place (e.g., keyboard shortcut or UI toggle)
- Each turret type uses its corresponding provided prefab
- Both turret types cost coins to place (may differ per type)

Story 8: Creep Variety

> *As a player, I face different types of creeps with varying difficulty.*

Acceptance Criteria:

- Two creep types exist: Small and Big
- Small creeps are faster with lower HP
- Big creeps are slower with higher HP
- Each type uses its corresponding provided prefab (Creep-small, Creep-big)
- Creep attributes (speed, HP, coin reward) are defined in data and easy to extend

---

Story 9: Wave System

> *As a player, I face successive waves of creeps, and I win if I survive them all.*

Acceptance Criteria:

- Creeps arrive in defined waves

- Each wave specifies which creep types spawn, how many, and at what intervals
- A wave is considered cleared when all its creeps are destroyed or have reached the base
- The next wave starts after the current wave is cleared (with optional delay)
- After all waves are cleared with the Base still alive, the WinPopup is displayed
- Wave definitions are data-driven and easy to add/modify

Story 10: Game Reset

> *As a player, I can restart the game after winning or losing without leaving Play Mode.*

Acceptance Criteria:

- A restart button/option is available on both the WinPopup and LosePopup
- Restarting resets all game state: Base health, coins, wave progress, all spawned creeps destroyed, all placed turrets removed
- The game returns to its initial starting state and begins again
- No residual state from the previous session affects the new game
- Works entirely within a single Play Mode session (no editor stop/start required)