Lab 5 (Unity): Goal-Oriented Action Planning (GOAP) Guard

Focus: Build a small GOAP system that can **plan**, **execute**, and **replan** in a live Unity scene.
As seen in Lecture 5, we'll use concepts such as: **Goals + World State + Actions + Planner + Executor**.

You'll reuse skills from earlier labs:

- **NavMeshAgent movement & simple NPC setup** (Lab 1)
- **Search-as-planning intuition (A* → now in state space)** (Lab 2)
- **Perception / "service-like" sensing updates** (Lab 4)
- Optional: smoothing / movement feel via steering ideas (Lab 3)

---

1) Learning goals

By the end of Lab 5, you should be able to:

1. **Represent a world state** as a set of facts the planner can reason about.
2. **Define GOAP actions** with:
    - Preconditions (what must be true)
    - Effects (what becomes true/false)
    - Cost (how "expensive" the action is)
3. **Implement a planner** (Dijkstra, which is A* but simpler, is enough) that searches the cheapest action sequence (like A*, but nodes are states).
4. **Execute the plan in the real scene** using NavMeshAgent, and **replan** if the world changes or an action fails.

---

2) Prerequisites

You should be comfortable with:

- Unity scene setup, prefabs, inspector wiring
- Basic C# classes, lists, enums
- NavMeshSurface + NavMeshAgent

You do **not** need to have completed Lab 1/2/3/4, but reusing parts of them may save you time.

---

3) Scenario: “The GOAP Guard” (simple but real)

You will build a prototype with primitive objects:

- **Player** (capsule) moves with WASD.
- **Guard** (capsule) runs GOAP.
- A **Weapon pickup** object exists somewhere in the level.
- A **Tag zone**: the guard “handles the intruder” by reaching the player and “tagging” them (no combat systems needed).

The behavior we want

- If the guard sees the player:
    - If it **doesn’t have a weapon**, it should **go get the weapon**, then **go tag the player**.
    - If it **already has a weapon**, it should **go tag the player** immediately.
- If the guard does **not** see the player:
    - It should **patrol** between waypoints.

This forces planning, because the “best next step” depends on state (facts).

---

4) Project & scene setup (Unity Editor)

Step 0: Choose project approach

Option A: New project

1. Create a new Unity 3D Core (or URP) project.
2. Create a scene: `Lab5_GOAP`.

Option B: Reuse Lab 4 project

- Add a new scene `Lab5_GOAP` and keep it separate. (Recommended, because you already have NavMesh + player patterns.)

Step 1: Packages

Ensure you have installed **AI Navigation** (NavMeshSurface, NavMeshAgent)

Step 2: Build a tiny test level + bake NavMesh

1. Add a Plane (floor).
2. Add a few Cubes as walls/obstacles.
3. Add a `NavMeshSurface` component to a GameObject (often `NavMeshRoot`), set it to collect render meshes, then **Bake**.

Step 3: Create Player

1. Add a Capsule named `Player`.
2. Add `SimplePlayerController.cs` (Appendix) to move with WASD.

Step 4: Create Guard

1. Add a Capsule named `Guard`.
2. Add `NavMeshAgent` component.
3. Add the scripts:
    - `GuardSensors.cs` (perception)
    - `GoapAgent.cs` (planner + executor)

Step 5: Create world objects

1. Create a Cube named `WeaponPickup`.
2. Place it somewhere reachable on the NavMesh.
3. Create 3–5 empty GameObjects as patrol waypoints: `WP_0...WP_4`.

---

5) Core theory reminders

5.1 Facts (world state)

Facts are booleans like:

- `SeesPlayer`
- `HasWeapon`
- `AtWeapon`
- `AtPlayer`
- `PatrolStepDone`
- `PlayerTagged`

The planner searches **state space**:

- **Nodes** = world states (sets of facts)
- **Edges** = actions that transform states (effects)

  This mirrors Lab 2's "nodes/edges/costs" thinking, just not in geometry.

5.2 Actions (STRIPS-like)

Each GOAP action has:

- **Preconditions**: facts that must already be true for the planner to consider this action.
- **Effects**: facts that become true/false *in the planner's model* if the action succeeds.
- **Cost**: how "expensive" the action is, used to choose between multiple valid plans.

Important modeling note (continuous behaviors):
Some behaviors (like "patrolling") are *continuous* and don't naturally become "true forever." In this lab, the patrol goal is modeled as a **small, concrete completion event**: the guard must successfully perform a patrol step (reach the next waypoint). We represent this as a fact named:

- `PatrolStepDone` = "the agent completed one patrol step"

This is only used for **goal satisfaction**. After that step succeeds, the plan ends, and if the player is still not seen, the agent will plan another patrol step. This "replan-to-continue" loop is intentional in this lab and keeps the GOAP loop visible and debuggable.

---

6) Lab tasks

Check the Appendix for reference code.

Part A: Implement the GOAP data model

Task A1: Define facts

Create an enum `GoapFact` with the facts listed above (you may add more).

Task A2: World state as a bitmask (no hand-waving)

We will represent state as a `ulong` bitmask:

- If bit `i` is 1 → fact `i` is true
- If bit `i` is 0 → fact `i` is false/unknown (we'll treat it as false here)

Why: planners need fast equality/hashing to avoid revisiting states.

Implement:

- `bool Has(GoapFact f)`
- `GoapState With(GoapFact f)` (returns new state with bit set)
- `GoapState Without(GoapFact f)` (returns new state with bit cleared)
- `bool Satisfies(ulong goalMask)` (goalMask bits must all be present)

Part B: Define actions as components (planner + runtime)

You will implement actions as MonoBehaviours attached to the Guard.

Task B1: Create a `GoapActionBase`

It must contain:

- `float Cost`
- `ulong PreMask` (required facts)
- `ulong AddMask` (facts to add on success)
- `ulong DelMask` (facts to delete on success)
- Runtime hooks:
    - `bool CheckProcedural(GoapContext ctx)` (raycasts, object existence, etc.)
    - `void OnEnter(GoapContext ctx)`

- `GoapStatus Tick(GoapContext ctx)`
- `void OnExit(GoapContext ctx)`

This matches the distinction between declarative vs procedural conditions mentioned in Lecture 5.

Task B2: Implement these actions (at least)

1. **PatrolAction**
   - Preconditions: none (or `!SeesPlayer` handled by goal selection)
   - Effects: add `PatrolStepDone`
   - Runtime: move between waypoints using NavMeshAgent (like Lab 1 patrol logic)
2. **MoveToWeaponAction**
   - Preconditions: `WeaponExists`
   - Effects: add `AtWeapon`
   - Runtime: set NavMeshAgent destination to WeaponPickup and succeed when close.
3. **PickUpWeaponAction**
   - Preconditions: `AtWeapon`
   - Effects: add `HasWeapon` and **delete `AtWeapon`** (recommended in this lab, since after pickup you're no longer "at the weapon pickup" in a meaningful planner sense)
   - Runtime: "pick up" by disabling the weapon object.
4. **MoveToPlayerAction**
   - Preconditions: `SeesPlayer`
   - Effects: add `AtPlayer`

- Runtime: set destination to Player each tick and succeed when close.
- If player no longer seen → fail (forces replanning).

Note: `SeesPlayer` can change during execution (player hides / breaks LOS). That's expected. The action may fail, triggering replanning. In this lab we also throttle replanning slightly to avoid noisy plan spam when perception flickers.

5. **TagPlayerAction**
   - Preconditions: `HasWeapon` and `AtPlayer`
   - Effects: add `PlayerTagged`
   - Runtime: log "Tagged intruder!" and maybe set a cooldown.

Task B3: Add a `GoapContext`

Note (important modeling point): `GoapContext` can contain mutable runtime variables (like `PatrolIndex`) that **do not belong in the planner's world model**, because they do not affect which plan is valid, only *how an action executes*. This is a concrete example of "runtime state that isn't planner-relevant."

Implement a small class/struct that holds references needed by actions:

- `NavMeshAgent agent`
- `Transform player`
- `Transform weaponPickup`
- `Transform[] patrolWaypoints`
- `GuardSensors sensors` (so actions can query perception)

Part C: Implement the planner (Dijkstra)

You will implement a planner using **Dijkstra search** in *state space*. This is essentially **A*** with the heuristic set to **0** (i.e., no heuristic guidance). The structure is the same as Lab 2 (open list, costs, came-from), but now:

- **Nodes = world states** (sets of facts)

- **Edges = actions** (apply effects to create successor states)

This keeps the planner conceptually aligned with the A* work you did earlier, but simpler to implement correctly.

Task C1: Planner inputs/outputs

Inputs:

- Start state (current facts)
- Goal mask (facts that must be true)
- List of actions (with pre/add/del/cost)

Output:

- A list/queue of actions (the plan), cheapest total cost.

Task C2: Implement Dijkstra cleanly

Use:

- `openList`: states to explore
- `costSoFar[state]`
- `cameFrom[state] = (prevState, actionUsed)`

Stop when you pop a state that satisfies the goal.

This is basically the same “search structure” you built in Lab 2, just with action-transitions instead of grid neighbors.

Part D: Implement the GOAP agent (goal selection + execution + replanning)

Task D1: Sensing updates → current facts

Create `GuardSensors.cs` similar in spirit to Lab 4: update `SeesPlayer` based on distance + optional raycast LOS.

Your `GoapAgent` should build a current `GoapState` each frame (or at a fixed tick rate), from:

- Sensors (`SeesPlayer`)
- Inventory (`HasWeapon`)
- World (`WeaponExists` if the weapon object is active)
- Execution markers (`AtWeapon`, `AtPlayer`, etc. can be "owned" by actions)

Task D2: Goal selection (no magic)

Implement this exact rule set:

- If `SeesPlayer == true` → **Goal = `PlayerTagged`**
- Else → **Goal = `PatrolStepDone`**

(Yes, it's simple. The complexity is in planning + execution, not in goal selection yet.)

Task D3: Plan execution

Maintain:

- `Queue<GoapActionBase> currentPlan`
- `GoapActionBase currentAction`

Execution loop:

1. If no plan → request plan from planner.
2. If currentAction is null → dequeue next action and call `OnEnter`.
3. Tick action each Update:
    - If `Running` → keep ticking
    - If `Success` → apply action effects to your agent's "owned" facts; `OnExit`; go next

- If `Failure` → invalidate plan and replan

Task D4: Replanning triggers (minimum)

Replan when:

- An action returns `Failure`
- The goal changes (player seen vs not seen)
- A “key fact” changes (weapon disappears, player disappears, etc.)

In Lecture 5 we explicitly expect replanning and plan invalidation as normal behavior.

Extra: Add a tiny debug HUD

Create a script GoapDebugHUD.cs, attach it to the Guard (or any object), then drag the GoapAgent reference in the inspector. Add debug helper functions to GoapAgent.cs (Appendix 7).

---

7) Testing checklist

You should be able to press Play and show:

1. If player is far away:

- Guard follows waypoint patrol (NavMesh) like earlier labs.

2. If player comes into detection range / LOS:

- Guard logs a new goal (`PlayerTagged`)

- Guard prints the chosen plan (in order), e.g.
  `MoveToWeapon → PickUpWeapon → MoveToPlayer → TagPlayer`
  or, if already armed:
  `MoveToPlayer → TagPlayer`

3. If player breaks LOS mid-plan:

- At least one action should fail

- The plan should invalidate and replan (or switch to patrol goal)

---

8) Hand-in

Same idea as earlier labs: submit either:

- a repo link in a .txt-file, or
- a zip/project link, or
- relevant code + short instructions to run (prefer repo).

Feel free to include in your repo's README:

- How to run the scene
- What facts/actions you implemented
- A short note describing one replanning scenario you tested

---

9) Common mistakes

- **Forgetting to “apply effects” only on action success.**
  Don’t update facts when the action starts—update when it completes successfully.

- **Planning with facts that are not stable.**
  `SeesPlayer` can change every frame. That’s fine, but then you must handle action failure and replanning.

- **Mixing declarative vs procedural logic.**
  Preconditions/effects must be *planner-visible*. Raycasts, reachability, and “is the object actually there” are procedural checks.
  Lecture 5 Preparations - GOAP

- **Infinite replanning loops.**
  If an action always fails procedurally (e.g., weapon behind a wall / unreachable), your agent can spam replans. Fix by:

  - adding a cooldown for replanning
  - marking some actions temporarily invalid
  - improving procedural checks

---

10) Reflection questions (few sentences each)

1. In your implementation, what are the **nodes** and **edges** in the search space? How is this similar/different to Lab 2?

2. Give one example where your action model could “lie” (effects claim success but the world disagrees). What happens?

3. Which parts of your system are **declarative** vs **procedural**?

4. When did your system replan, and why was that the correct moment?

Appendix — Reference implementation (Unity/C#)

> This code is intentionally minimal-but-working. You should still read it and understand how facts/actions/planning/execution connect to each other.

A.1 `SimplePlayerController.cs`

If you have not yet set up the modern Unity Input System, you may need to follow this:

Unity Editor setup (Input System)

1. **Install**: `Window → Package Manager → Input System` (install)
2. **Enable** (if prompted): set **Active Input Handling** to **Input System Package** (or **Both**) in `Edit → Project Settings → Player → Other Settings → Active Input Handling`, then restart Unity.
3. Create an Input Actions asset:
   `Assets → Create → Input Actions` → name it `PlayerControls`
4. Open `PlayerControls` and add:
   - **Action Map**: `Player`
   - **Action**: `Move`
     - Action Type: **Value**
     - Control Type: **Vector2**
     - Add Binding: **2D Vector Composite**
       - Up: `W`
       - Down: `S`
       - Left: `A`
       - Right: `D`

        - (Optional) Add another binding: **Gamepad Left Stick**

5. Select your `Player` GameObject and add:
    - **PlayerInput** component
        - Actions: assign `PlayerControls`
        - Default Map: `Player`
        - Behavior: **Send Messages** (simplest for this lab)

SimplePlayerController.cs (Input System)

```
using UnityEngine;
using UnityEngine.InputSystem;

public class SimplePlayerController : MonoBehaviour
{
    [Header("Movement")]
    public float speed = 5f;

    private Vector2 _moveInput; // from Input System action "Move"

    void Update()
    {
        Vector3 dir = new Vector3(_moveInput.x, 0f, _moveInput.y);

        if (dir.sqrMagnitude > 1f)
            dir.Normalize();

        transform.position += dir * speed * Time.deltaTime;

        if (dir.sqrMagnitude > 0.001f)
            transform.forward = dir;
    }


    // PlayerInput (Send Messages) calls this automatically for an
action named "Move"
    // Signature must match: void On<ActionName>(InputValue value)
```

```
    public void OnMove(InputValue value)
    {
        _moveInput = value.Get<Vector2>();
    }
}
```

A.2 GuardSensors.cs

```
using UnityEngine;

public class GuardSensors : MonoBehaviour
{
    public Transform player;
    public float viewRange = 10f;
    public LayerMask occluders = ~0; // everything by default
    public bool useLineOfSightRaycast = true;

    public bool SeesPlayer { get; private set; }

    void Update()
    {
        SeesPlayer = false;
        if (player == null) return;

        float dist = Vector3.Distance(transform.position,
player.position);
        if (dist > viewRange) return;

        if (!useLineOfSightRaycast)
        {
            SeesPlayer = true;
            return;
        }

        Vector3 origin = transform.position + Vector3.up * 0.5f;
        Vector3 target = player.position + Vector3.up * 0.5f;
        Vector3 dir = (target - origin);
        float len = dir.magnitude;

        if (len < 0.001f) { SeesPlayer = true; return; }
```

```
        if (Physics.Raycast(origin, dir / len, out RaycastHit hit,
len, occluders))
        {
            // Sees player only if the first thing hit is the player
            if (hit.transform == player) SeesPlayer = true;
        }
    }
}
```

A.3 `GoapCore.cs` (facts, state, planner)

```
using System;
using System.Collections.Generic;

public enum GoapFact
{
    SeesPlayer = 0,
    WeaponExists = 1,
    HasWeapon = 2,
    AtWeapon = 3,
    AtPlayer = 4,
    PatrolStepDone = 5,
    PlayerTagged = 6,
}

public readonly struct GoapState : IEquatable<GoapState>
{
    public readonly ulong Bits;

    public GoapState(ulong bits) => Bits = bits;

    public bool Has(GoapFact f) => (Bits & (1UL << (int)f)) != 0;

    public GoapState With(GoapFact f) => new GoapState(Bits | (1UL
<< (int)f));
    public GoapState Without(GoapFact f) => new GoapState(Bits &
~(1UL << (int)f));

    public bool Satisfies(ulong goalMask) => (Bits & goalMask) ==
goalMask;
```

```
    public bool Equals(GoapState other) => Bits == other.Bits;
    public override bool Equals(object obj) => obj is GoapState s &&
Equals(s);
    public override int GetHashCode() => Bits.GetHashCode();
}

public static class GoapBits
{
    public static ulong Mask(params GoapFact[] facts)
    {
        ulong m = 0;
        foreach (var f in facts) m |= 1UL << (int)f;
        return m;
    }
}

public sealed class GoapPlanResult
{
    public readonly List<GoapActionBase> Actions = new();
    public float TotalCost;
}

public static class GoapPlanner
{
    private struct CameFrom
    {
        public GoapState Prev;
        public GoapActionBase Action;
        public bool HasPrev;
    }

    // Dijkstra in state-space (small action sets => simple open
list is fine)
    public static GoapPlanResult Plan(GoapState start, ulong
goalMask, List<GoapActionBase> actions)
    {
        var open = new List<GoapState> { start };

        var cost = new Dictionary<GoapState, float> { [start] = 0f
};
        var came = new Dictionary<GoapState, CameFrom>();
```

```
        while (open.Count > 0)
        {
            // pick lowest-cost state (O(n); OK for small demos)
            int bestIdx = 0;
            float bestCost = cost[open[0]];
            for (int i = 1; i < open.Count; i++)
            {
                float c = cost[open[i]];
                if (c < bestCost) { bestCost = c; bestIdx = i; }
            }

            var current = open[bestIdx];
            open.RemoveAt(bestIdx);

            if (current.Satisfies(goalMask))
                return Reconstruct(current, came, cost[current]);

            foreach (var a in actions)
            {
                if (!a.CanApplyTo(current)) continue;

                var next = a.ApplyTo(current);
                float newCost = cost[current] + a.cost;

                if (!cost.TryGetValue(next, out float old) ||
newCost < old)
                {
                    cost[next] = newCost;
                    came[next] = new CameFrom { Prev = current,
Action = a, HasPrev = true };
                    if (!open.Contains(next)) open.Add(next);
                }
            }
        }

        return null;
    }

    private static GoapPlanResult Reconstruct(GoapState goalState,
Dictionary<GoapState, CameFrom> came, float totalCost)
```

```
    {
        var result = new GoapPlanResult { TotalCost = totalCost };

        var cur = goalState;
        while (came.TryGetValue(cur, out var step) && step.HasPrev)
        {
            result.Actions.Add(step.Action);
            cur = step.Prev;
        }

        result.Actions.Reverse();
        return result;
    }
}
```

A.4 `GoapActionBase.cs` (action definition + runtime hooks)

```
using UnityEngine;
using UnityEngine.AI;

public enum GoapStatus { Running, Success, Failure }

public class GoapContext
{
    public NavMeshAgent Agent;
    public Transform Player;
    public Transform Weapon;
    public Transform[] PatrolWaypoints;
    public GuardSensors Sensors;
    public int PatrolIndex;
}

public abstract class GoapActionBase : MonoBehaviour
{
    [Header("GOAP (planner-visible)")]
    public string actionName = "Action";
    public float cost = 1f;

    // planner-visible masks (pre/add/del)
```

```
    public ulong preMask;
    public ulong addMask;
    public ulong delMask;

    // Runtime (procedural)
    public virtual bool CheckProcedural(GoapContext ctx) => true;
    public virtual void OnEnter(GoapContext ctx) { }
    public abstract GoapStatus Tick(GoapContext ctx);
    public virtual void OnExit(GoapContext ctx) { }

    // Planner helpers
    public bool CanApplyTo(GoapState s) => (s.Bits & preMask) ==
preMask;
    public GoapState ApplyTo(GoapState s)
    {
        ulong bits = s.Bits;
        bits &= ~delMask;
        bits |= addMask;
        return new GoapState(bits);
    }
}
```

A.5 `GoapAgent.cs` (goal selection + planning + execution)

Replace the whole file with this updated version:

```
using System.Collections.Generic;
using System.Text;
using UnityEngine;
using UnityEngine.AI;

public class GoapAgent : MonoBehaviour
{
    [Header("Scene refs")]
    public GuardSensors sensors;
    public Transform player;
    public Transform weaponPickup;
    public Transform[] patrolWaypoints;
```

```
    [Header("Debug")]
    public bool logPlans = true;

    [Header("Planning")]
    [Tooltip("Minimum seconds between replans (prevents spam when
facts flicker).")]
    public float minSecondsBetweenReplans = 0.20f;

    private float _nextAllowedReplanTime = 0f;

    private NavMeshAgent _agent;
    private GoapContext _ctx;

    private List<GoapActionBase> _allActions;
    private Queue<GoapActionBase> _plan;
    private GoapActionBase _currentAction;

    // “Owned” facts: memory/execution facts (e.g., HasWeapon,
AtWeapon, AtPlayer, PatrolStepDone, PlayerTagged)
    // Sensor/world facts (SeesPlayer, WeaponExists) are refreshed
each tick.
    private ulong _ownedFactsBits = 0;

    void Awake()
    {
        _agent = GetComponent<NavMeshAgent>();

        _ctx = new GoapContext
        {
            Agent = _agent,
            Player = player,
            Weapon = weaponPickup,
            PatrolWaypoints = patrolWaypoints,
            Sensors = sensors,
            PatrolIndex = 0
        };

        _allActions = new
List<GoapActionBase>(GetComponents<GoapActionBase>());
    }
```

```
    void Update()
    {
        GoapState current = BuildCurrentState();
        ulong goalMask = SelectGoalMask(current);

        // If we have no plan, request one (throttled).
        if ((_plan == null || _plan.Count == 0) && Time.time >=
_nextAllowedReplanTime)
        {
            MakePlan(current, goalMask);
        }

        if (_plan == null || _plan.Count == 0) return;

        // Start next action if needed
        if (_currentAction == null)
        {
            _currentAction = _plan.Dequeue();

            // Procedural check at runtime (not planner-visible)
            if (!_currentAction.CheckProcedural(_ctx))
            {
                InvalidatePlan(throttle: true);
                return;
            }

            _currentAction.OnEnter(_ctx);
        }

        var status = _currentAction.Tick(_ctx);

        if (status == GoapStatus.Running) return;

        if (status == GoapStatus.Success)
        {
            // Apply effects only on success
            ApplyActionEffectsToOwnedFacts(_currentAction);
            _currentAction.OnExit(_ctx);
            _currentAction = null;
            return;
        }
```

```
        // Failure: action did not complete; invalidate and replan
(throttled)
        _currentAction.OnExit(_ctx);
        _currentAction = null;
        InvalidatePlan(throttle: true);
    }

    private GoapState BuildCurrentState()
    {
        ulong bits = _ownedFactsBits;

        // Determine owned HasWeapon first (used to interpret
WeaponExists as "pickup available to THIS agent")
        bool hasWeapon = (bits & GoapBits.Mask(GoapFact.HasWeapon))
!= 0;

        // Sensor-driven fact (fresh each tick)
        if (sensors != null && sensors.SeesPlayer) bits |=
GoapBits.Mask(GoapFact.SeesPlayer);
        else bits &= ~GoapBits.Mask(GoapFact.SeesPlayer);

        // World-driven fact (fresh each tick)
        // Interpret WeaponExists as "pickup available to pick up".
        // If the agent already has a weapon, treat WeaponExists as
false for planning purposes.
        bool pickupActive = weaponPickup != null &&
weaponPickup.gameObject.activeInHierarchy;
        bool weaponAvailable = pickupActive && !hasWeapon;

        if (weaponAvailable) bits |=
GoapBits.Mask(GoapFact.WeaponExists);
        else bits &= ~GoapBits.Mask(GoapFact.WeaponExists);

        return new GoapState(bits);
    }

    private ulong SelectGoalMask(GoapState current)
    {
        // Simple rule set:
        // - If player seen: goal is to tag the player
```

```
        // - Else: goal is to complete one patrol step
        if (current.Has(GoapFact.SeesPlayer))
            return GoapBits.Mask(GoapFact.PlayerTagged);

        return GoapBits.Mask(GoapFact.PatrolStepDone);
    }

    private void MakePlan(GoapState current, ulong goalMask)
    {
        var res = GoapPlanner.Plan(current, goalMask, _allActions);
        if (res == null)
        {
            if (logPlans) Debug.LogWarning("GOAP: No plan found.");
            _plan = null;
            return;
        }

        _plan = new Queue<GoapActionBase>(res.Actions);

        if (logPlans)
        {
            var sb = new StringBuilder();
            sb.AppendLine($"GOAP Plan (cost {res.TotalCost:0.0}):");
            foreach (var a in res.Actions) sb.AppendLine($"-
{a.actionName} (cost {a.cost:0.0})");
            Debug.Log(sb.ToString());
        }
    }

    private void InvalidatePlan(bool throttle)
    {
        _plan = null;
        _currentAction = null;

        if (throttle)
            _nextAllowedReplanTime = Time.time +
minSecondsBetweenReplans;
    }

    private void ApplyActionEffectsToOwnedFacts(GoapActionBase a)
    {
```

```
            _ownedFactsBits &= ~a.delMask;
            _ownedFactsBits |= a.addMask;
        }
    }
```

Note:

- PatrolStepDone is explicitly a one-step goal marker (not “patrolling forever”).
- The “success → replan again to keep patrolling” pattern is explained in code/comments and in the text.
- Replanning is throttled slightly so flickering SeesPlayer doesn’t look buggy.
- WeaponExists is interpreted as “pickup available to pick up,” avoiding the edge case where the pickup reactivates but the guard already has a weapon.

A.6 Actions (attach these to `Guard`)

```
Replace the whole file with:

using UnityEngine;

public class PatrolAction : GoapActionBase
{
    public float arriveDistance = 0.7f;

    void Reset()
    {
        actionName = "Patrol (One Step)";
        cost = 2f;

        // No planner preconditions; goal selection handles “patrol
only when not chasing”.
        preMask = 0;

        // This is intentionally a "one-step completion" fact.
        addMask = GoapBits.Mask(GoapFact.PatrolStepDone);
        delMask = 0;
    }

    public override void OnEnter(GoapContext ctx)
```

```
    {
        if (ctx.PatrolWaypoints == null ||
ctx.PatrolWaypoints.Length == 0) return;

ctx.Agent.SetDestination(ctx.PatrolWaypoints[ctx.PatrolIndex].positi
on);
    }

    public override GoapStatus Tick(GoapContext ctx)
    {
        // If the player appears, we want to stop patrolling and
replan for the chase goal.
        if (ctx.Sensors != null && ctx.Sensors.SeesPlayer)
            return GoapStatus.Failure;

        if (ctx.PatrolWaypoints == null ||
ctx.PatrolWaypoints.Length == 0)
            return GoapStatus.Failure;

        if (ctx.Agent.pathPending) return GoapStatus.Running;

        if (ctx.Agent.remainingDistance <= arriveDistance)
        {
            // “Success” here means: completed ONE patrol step
(reached current waypoint).
            // We increment the patrol index here, but we do NOT set
a new destination, because this action is ending.
            // The next Patrol action’s OnEnter() will set the new
destination for the next waypoint.
            ctx.PatrolIndex = (ctx.PatrolIndex + 1) %
ctx.PatrolWaypoints.Length;

            return GoapStatus.Success;
        }

        return GoapStatus.Running;
    }
}
```

MoveToWeaponAction.cs

```
using UnityEngine;

public class MoveToWeaponAction : GoapActionBase
{
    public float arriveDistance = 0.7f;

    void Reset()
    {
        actionName = "Move To Weapon";
        cost = 1f;
        preMask = GoapBits.Mask(GoapFact.WeaponExists);
        addMask = GoapBits.Mask(GoapFact.AtWeapon);
        delMask = 0;
    }

    public override bool CheckProcedural(GoapContext ctx)
    {
        return ctx.Weapon != null &&
ctx.Weapon.gameObject.activeInHierarchy;
    }

    public override void OnEnter(GoapContext ctx)
    {
        ctx.Agent.SetDestination(ctx.Weapon.position);
    }

    public override GoapStatus Tick(GoapContext ctx)
    {
        if (ctx.Weapon == null ||
!ctx.Weapon.gameObject.activeInHierarchy)
            return GoapStatus.Failure;

        if (ctx.Agent.pathPending) return GoapStatus.Running;

        if (ctx.Agent.remainingDistance <= arriveDistance)
            return GoapStatus.Success;

        return GoapStatus.Running;
    }
}
```

PickUpWeaponAction.cs

```
using UnityEngine;

public class PickUpWeaponAction : GoapActionBase
{
    void Reset()
    {
        actionName = "Pick Up Weapon";
        cost = 1f;
        preMask = GoapBits.Mask(GoapFact.AtWeapon);
        addMask = GoapBits.Mask(GoapFact.HasWeapon);
        delMask = GoapBits.Mask(GoapFact.AtWeapon); // recommended
    }

    public override bool CheckProcedural(GoapContext ctx)
    {
        return ctx.Weapon != null &&
ctx.Weapon.gameObject.activeInHierarchy;
    }

    public override GoapStatus Tick(GoapContext ctx)
    {
        if (ctx.Weapon == null ||
!ctx.Weapon.gameObject.activeInHierarchy)
            return GoapStatus.Failure;

        ctx.Weapon.gameObject.SetActive(false);
        return GoapStatus.Success;
    }
}
```

MoveToPlayerAction.cs

```
using UnityEngine;

public class MoveToPlayerAction : GoapActionBase
{
    public float arriveDistance = 1.2f;
```

```
    void Reset()
    {
        actionName = "Move To Player";
        cost = 1f;
        preMask = GoapBits.Mask(GoapFact.SeesPlayer);
        addMask = GoapBits.Mask(GoapFact.AtPlayer);
        delMask = 0;
    }

    public override bool CheckProcedural(GoapContext ctx)
    {
        return ctx.Player != null;
    }

    public override GoapStatus Tick(GoapContext ctx)
    {
        if (ctx.Player == null) return GoapStatus.Failure;

        // If we lose sight, fail → triggers replanning (as
required)
        if (ctx.Sensors != null && !ctx.Sensors.SeesPlayer)
            return GoapStatus.Failure;

        ctx.Agent.SetDestination(ctx.Player.position);

        if (ctx.Agent.pathPending) return GoapStatus.Running;

        if (ctx.Agent.remainingDistance <= arriveDistance)
            return GoapStatus.Success;

        return GoapStatus.Running;
    }
}
```

TagPlayerAction.cs

```
using UnityEngine;

public class TagPlayerAction : GoapActionBase
{
    void Reset()
```

```
    {
        actionName = "Tag Player";
        cost = 1f;
        preMask = GoapBits.Mask(GoapFact.HasWeapon,
GoapFact.AtPlayer);
        addMask = GoapBits.Mask(GoapFact.PlayerTagged);
        delMask = 0;
    }

    public override GoapStatus Tick(GoapContext ctx)
    {
        Debug.Log("GOAP: Tagged intruder!");
        return GoapStatus.Success;
    }
}
```

A.7 GOAP DebugHUD (attach/add to `Guard`)

`GoapDebugHUD.cs`

```
using System.Text;
using UnityEngine;

public class GoapDebugHUD : MonoBehaviour
{
    public GoapAgent agent;

    [Header("HUD")]
    public bool show = true;
    public Vector2 screenOffset = new Vector2(10, 10);

    GUIStyle _style;

    void Awake()
    {
        _style = new GUIStyle(GUI.skin.label)
        {
            fontSize = 14,
            richText = true
        };
```

```
    }

    void OnGUI()
    {
        if (!show || agent == null) return;

        var sb = new StringBuilder();
        sb.AppendLine("<b>GOAP Debug</b>");
        sb.AppendLine(agent.GetDebugString());

        GUI.Label(new Rect(screenOffset.x, screenOffset.y, 600,
800), sb.ToString(), _style);
    }
}
```

Add these debug helpers to `GoapAgent.cs`

Try pasting the following **inside the `GoapAgent` class** (anywhere, probably near the bottom). This exposes clean debug text without duplicating logic in the HUD.

```
public string GetDebugString()
{
    var s = BuildCurrentState();

    var sb = new System.Text.StringBuilder();

    sb.AppendLine($"Goal: {GoalMaskToString(SelectGoalMask(s))}");
    sb.AppendLine($"Current Action: {(_currentAction != null ?
_currentAction.actionName : "(none)")}");

    sb.AppendLine("Facts:");
    foreach (GoapFact f in System.Enum.GetValues(typeof(GoapFact)))
        sb.AppendLine($"- {f}: {(s.Has(f) ? "true" : "false")}");

    sb.AppendLine("Plan:");
    if (_plan == null || _plan.Count == 0)
    {
        sb.AppendLine("- (none)");
    }
    else
    {
        foreach (var a in _plan)
```

```
            sb.AppendLine($"- {a.actionName}");
    }

    return sb.ToString();
}

private string GoalMaskToString(ulong goalMask)
{
    var sb = new System.Text.StringBuilder();
    bool first = true;

    foreach (GoapFact f in System.Enum.GetValues(typeof(GoapFact)))
    {
        ulong bit = 1UL << (int)f;
        if ((goalMask & bit) != 0)
        {
            if (!first) sb.Append(", ");
            sb.Append(f);
            first = false;
        }
    }

    return first ? "(none)" : sb.ToString();
}
```

Note: This calls `BuildCurrentState()` for debug display (fine for this lab). If you later optimize, you can cache the last computed state each Update.