Cél

Rugalmas, bővíthető, többdimenziós ügynök-viselkedés WorldSim-hez (MonoGame/C#), neurális háló nélkül, későbbi LLM/chatbot integrációs opcióval.

Nagykép (komponálható hibrid)

```
[Sensing/Perception]
   └► Factual Events (észlelések)
         └► BLACKBOARD (rövid táv) + MEMORY (hosszú táv)
               ├► NEEDS/EMOTIONS/TRAITS (állapot + paraméterek)
               └► WORLD MODEL (térkép, tárgyak, kapcsolatok)

[Goals]
   └► Utility AI (szempontok/considerations → pontszám)
         └► Legjobb(ak) cél kiválasztása + tehetetlenség/cooldown

[Planner]
   ├► GOAP (A* a cselekvések terén)
   └►/vagy HTN (hierarchikus taskok rutinokra)

[Executor]
   └► Behavior Tree (reaktív futtatás, megszakítások, fallback)
         └► Actions (atomi/mikro: MoveTo, PickUp, Eat, Talk...)

[Motor layer]
   ├► Pathfinding (rács/navmesh)
   ├► Steering (Reynolds/social forces)
   └► Animation/Timing (akciók időzítése)

[Infra]
   ├► EventBus (pub/sub)
   ├► Telemetria/Debugger (BT/GOAP vizualizáció)
   └► Data-driven config (JSON/YAML)
```

Miért ez a mix? - **Utility AI** nagyon jól skálázik sok, egymásnak feszülő motivációra (éhség, energia, társas, kötelesség, pénz, veszély stb.). - **GOAP** jól kezeli a több-lépéses célokat ("Szerezz ételt" → Pénz? → Bolt → Vásárlás → Főzés → Evés). - **BT** reaktív, könnyen megszakítja/folytatja az akciókat, és jó végrehajtási kontrollt ad. - **Blackboard** + **EventBus** lazán csatolt modulok, könnyű bővítés.

Adatmodell – röviden

- **Needs** (0..1): Hunger, Energy, Hygiene, Social, Safety, Fun, WealthPressure...
- **Emotions/Mood** (pl. PAD: Pleasure–Arousal–Dominance, −1..+1)

- **Traits** (pl. conscientiousness, riskTolerance, agreeableness, introversion…)
- **Skills/Stats** (Cooking, Strength, Speech…)
- **Roles** (Worker, Student, Parent…)
- **World facts** (helyek, tárgyak, emberek, kapcsolatok, affordance-ok)

Data-driven példa (JSON)

Goal – „EatUntilComfortable”

```
{
  "id": "eat",
  "targetUtility": 0.85,
  "considerations": [
    {"type": "NeedCurve", "need": "Hunger", "slope": 2.5, "shift": 0.2},
    {"type": "TraitBias", "trait": "Conscientiousness", "weight": 0.2}
  ],
  "cooldownSec": 60,
  "minInertiaSec": 20
}
```

Action – „CookMeal” (GOAP)

```
{
  "id": "CookMeal",
  "pre": {"HasRawFood": true, "Near": "Kitchen"},
  "eff": {"Hunger": -0.4, "HasMeal": true},
  "cost": 4,
  "durationSec": 30
}
```

C# váz (interfészek + minták)

> **Megjegyzés:** Példák MonoGame-kompatibilis, tiszta C# formában. A környezeti integrációt (Update/Draw) a saját architektúrádhoz igazítsd. Minden sorban komment, hogy könnyen illeszthető legyen.

Blackboard

```
public record BBKey<T>(string Name); // típussafe kulcs

public interface IBlackboard // rövid távú, megosztott memória
{
    bool TryGet<T>(BBKey<T> key, out T value);
    void Set<T>(BBKey<T> key, T value);
```

```
}

public sealed class Blackboard : IBlackboard
{
    private readonly Dictionary<string, object> _data = new();

    public bool TryGet<T>(BBKey<T> key, out T value)
    {
        if (_data.TryGetValue(key.Name, out var obj) && obj is T t)
        { value = t; return true; }
        value = default!; return false;
    }

    public void Set<T>(BBKey<T> key, T value) => _data[key.Name] = value!;
}
```

Utility AI – szempontok + célpontozás

```
public interface IConsideration // egy szempont, 0..1
{
    float Score(Agent a, IBlackboard bb);
}

public sealed class NeedCurve : IConsideration
{
    public string Need; public float Slope = 2.0f; public float Shift =
0.0f; // paraméterezhető
    public NeedCurve(string need, float slope = 2f, float shift = 0f)
    { Need = need; Slope = slope; Shift = shift; }

    public float Score(Agent a, IBlackboard bb)
    {
        float v = a.Needs[Need]; // 0..1 éhség stb.
        // logisztikus jellegű görbe, finomítható lookup-kal
        float x = Math.Clamp((v - Shift) * Slope, -6f, 6f);
        return 1f / (1f + MathF.Exp(-x));
    }
}

public sealed class TraitBias : IConsideration
{
    public string Trait; public float Weight; // kis súly, hogy ne domináljon
    public TraitBias(string trait, float weight = 0.15f){ Trait = trait;
Weight = weight; }
    public float Score(Agent a, IBlackboard bb)
        => Math.Clamp(0.5f + (a.Traits[Trait] - 0.5f) * Weight, 0f, 1f);
}

public sealed class Goal
{
```

```
    public string Id = string.Empty;
    public List<IConsideration> Considerations = new();
    public float TargetUtility = 0.75f; // döntési küszöb
    public float CooldownSec = 30f;
    public float MinInertiaSec = 10f;
    private DateTime _lastPicked = DateTime.MinValue;

    public float Evaluate(Agent a, IBlackboard bb)
    {
        // Multiplikatív aggregáció, "compensation" faktorral (elkerüli a
mindent 1-re húzó összeget)
        float prod = 1f; float comp = 1f;
        foreach (var c in Considerations)
        {
            float s = Math.Clamp(c.Score(a, bb), 0.0001f, 1f);
            prod *= s; comp *= 0.9f + 0.1f * s; // kis kompenzáció
        }
        float u = MathF.Pow(prod, 1f / comp); // kiegyensúlyozottabb görbe

        // cooldown/inercia hatás
        var now = DateTime.UtcNow;
        float cooldownPenalty = (float)Math.Clamp(1.0 - (now -
_lastPicked).TotalSeconds / CooldownSec, 0, 1);
        return Math.Clamp(u * (1f - cooldownPenalty * 0.5f), 0f, 1f);
    }

    public void MarkPicked() => _lastPicked = DateTime.UtcNow;
}
```

GOAP – akciók és tervező (A* skeleton)

```
public sealed class WorldState : Dictionary<string, object> // egyszerű
baseline
{
    public WorldState Clone() => new WorldState(this);
}

public interface IGoapAction
{
    string Id { get; }
    IReadOnlyDictionary<string, object> Preconditions { get; }
    IReadOnlyDictionary<string, object> Effects { get; }
    float Cost { get; }
    float DurationSec { get; }
    bool Check(Agent a, WorldState s); // pl. hozzáférés/helyzet valós
ellenőrzés
    IEnumerator<bool> Execute(Agent a); // coroutine-szerű futtatás
}

public sealed class GoapPlanner
```

```
{
    public List<IGoapAction> Plan(WorldState start, Predicate<WorldState>
goalTest, IEnumerable<IGoapAction> actions)
    {
        // A* a worldstate-térben; heurisztika: h(s) ~ nem-teljesített
célfeltételek száma
        var open = new PriorityQueue<Node, float>();
        var startNode = new Node(start, null, null, g:0, h:Heuristic(start,
goalTest));
        open.Enqueue(startNode, startNode.F);
        var visited = new HashSet<string>();

        while (open.Count > 0)
        {
            var n = open.Dequeue();
            if (goalTest(n.State)) return Reconstruct(n);
            string hash = Hash(n.State);
            if (!visited.Add(hash)) continue;

            foreach (var act in actions)
            {
                if (!act.Check(null!, n.State)) continue; // agent hiányát
pótold
                if (!PreconditionsMet(n.State, act.Preconditions)) continue;
                var next = Apply(n.State, act.Effects);
                float g = n.G + act.Cost;
                float h = Heuristic(next, goalTest);
                var child = new Node(next, n, act, g, h);
                open.Enqueue(child, child.F);
            }
        }
        return new List<IGoapAction>(); // nincs terv

        // --- lokális függvények ---
        static bool PreconditionsMet(WorldState s,
IReadOnlyDictionary<string, object> pre)
        { foreach (var kv in pre) if (!s.TryGetValue(kv.Key, out var v) || !
Equals(v, kv.Value)) return false; return true; }
        static WorldState Apply(WorldState s, IReadOnlyDictionary<string,
object> eff)
        { var ns = s.Clone(); foreach (var kv in eff) ns[kv.Key] = kv.Value;
return ns; }
        static float Heuristic(WorldState s, Predicate<WorldState> goal) =>
goal(s) ? 0f : 1f; // csere: okosabb h()
        static string Hash(WorldState s) => string.Join("|", s.OrderBy(kv =>
kv.Key).Select(kv => kv.Key+"="+kv.Value));
        static List<IGoapAction> Reconstruct(Node n)
        { var path = new List<IGoapAction>(); while (n.Action != null) {
path.Add(n.Action); n = n.Parent!; } path.Reverse(); return path; }
    }
```

```
    private sealed class Node
    {
        public WorldState State; public Node? Parent; public IGoapAction?
Action; public float G; public float H; public float F => G + H;
        public Node(WorldState s, Node? p, IGoapAction? a, float g, float h)
{ State=s; Parent=p; Action=a; G=g; H=h; }
    }
}
```

Behavior Tree – végrehajtó

```
public enum NodeStatus { Success, Failure, Running }

public abstract class BTNode
{ public abstract NodeStatus Tick(Agent a, IBlackboard bb, float dt); }

public sealed class Sequence : BTNode
{
    private readonly BTNode[] _children; private int _i;
    public Sequence(params BTNode[] children){ _children = children; }
    public override NodeStatus Tick(Agent a, IBlackboard bb, float dt)
    {
        while (_i < _children.Length)
        {
            var s = _children[_i].Tick(a, bb, dt);
            if (s == NodeStatus.Running) return NodeStatus.Running;
            if (s == NodeStatus.Failure) { _i = 0; return
NodeStatus.Failure; }
            _i++;
        }
        _i = 0; return NodeStatus.Success;
    }
}

public sealed class Selector : BTNode
{
    private readonly BTNode[] _children; private int _i;
    public Selector(params BTNode[] children){ _children = children; }
    public override NodeStatus Tick(Agent a, IBlackboard bb, float dt)
    {
        while (_i < _children.Length)
        {
            var s = _children[_i].Tick(a, bb, dt);
            if (s == NodeStatus.Running) return NodeStatus.Running;
            if (s == NodeStatus.Success) { _i = 0; return
NodeStatus.Success; }
            _i++;
        }
        _i = 0; return NodeStatus.Failure;
    }
```

```
}

public sealed class Condition : BTNode
{
    private readonly Func<Agent, IBlackboard, bool> _pred;
    public Condition(Func<Agent, IBlackboard, bool> pred){ _pred = pred; }
    public override NodeStatus Tick(Agent a, IBlackboard bb, float dt)
        => _pred(a, bb) ? NodeStatus.Success : NodeStatus.Failure;
}

public sealed class DoAction : BTNode
{
    private readonly Func<Agent, IBlackboard, float, NodeStatus> _act;
    public DoAction(Func<Agent, IBlackboard, float, NodeStatus> act){ _act =
act; }
    public override NodeStatus Tick(Agent a, IBlackboard bb, float dt) =>
_act(a, bb, dt);
}
```

Ügynök fő ciklus (pszeudó)

```
// Update(dt):
PerceptionSystem.Update(a, bb, dt);           // szenzorok → blackboard
var goal = GoalSelector.PickBest(a, bb);      // Utility AI
if (Planner.NeedsReplan(a, goal))             // ha hiányzik/invalid a terv
    a.Plan = GoapPlanner.Plan(a.World, goal.Test, ActionLibrary.All);
BTExecutor.Tick(a, bb, dt);                   // végrehajtás + reaktivitás
MovementSystem.Update(a, dt);                 // pathfinding + steering
NeedsSystem.Decay(a, dt);                     // igények/mood változás
```

Mozgás (Motor layer) – rövid recept

1. **Pathfinding**: rácson A* (vagy navmesh), dinamikus akadályokkal (emberek).
2. **Steering**: arrive, avoid, separation/alignment (tömegben természetes mozgás).
3. **Scheduler**: párhuzamos mikro-akciók (pl. „MoveTo” + „LookAt”), prioritás és megszakítás BT-n keresztül.

Meta-kontroll (robosztusság)

- **Inercia**: ne váltogasson percenként célt; min. idő egy célon.
- **Cooldown**: túlhasznált célokra/akciókra.
- **Stochasticity**: kis zaj a pontszámokban (különböző egyedek).
- **Social rules**: "proximity etiquette" (távolság), sorban állás, ütközés elkerülés.
- **Time budget**: frame-limit a tervezőn (szétosztott keresés több frame-re).

LLM/Chatbot-készenlét (jövőbiztos)

- **Intent API**: minden cél/akció kap egy *ember-olvasható* leírást és paraméterezhető slotokat.
- **"Tool" réteg**: akciók mint eszközök (signature: name, args schema, pre/eff/cost). Egy LLM a későbbiekben javasolhat sorrendet, de a **Check/Execute** továbbra is determinisztikus, játékmotor-oldalon.
- **Narratív/Dialogue**: `TalkTo(target, topic)` akciók, ahol a topic-ot később LLM töltheti.

Bevezetési ütemterv

- **v0 (1–2 nap)**: Needs + Utility selector + pár BT akció (Eat/Sleep/Work/MoveTo).
- **v1 (1 hét)**: GOAP 5–10 akcióval (AcquireFood pipeline), cooldown/inercia, telemetria overlay.
- **v2**: HTN rutinok (napi rend), social steering, influence map.
- **v3**: LLM-hook (opcionális), dialógus-eszközök.

WorldSim – konkrét integráció (V0)

Alább egy **minimális, de azonnal hasznos** integráció: a jelenlegi `Person.Update(...)` véletlen job-választását kiváltjuk **Utility-alapú döntéssel**, és a random `Wander` helyett **célirányos lépegetést** adunk a legközelebbi erőforrás felé. (Később erre ráépítjük a teljes GOAP/BT/HFSM-et.)

1) Új fájl: `Simulation/AI/AgentBrain.cs`

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

namespace WorldSim.Simulation.AI
{
    // Magas szintű (V0) célok – később bővítjük (GOAP/BT/HFSM)
    public enum GoalId { Idle, GatherWood, BuildHouse }

    public sealed class AgentBrain
    {
        readonly Random _rng = new();

        // Egyszerű utility-értékelés: BuildHouse > GatherWood > Idle
        public GoalId DecideGoal(Person p, World w)
        {
            var home = p.Home;
            int colonyPop = w._people.Count(x => x.Home == home);
            int capacity = home.HouseCount * w.HouseCapacity;
            bool needHousing = colonyPop >= capacity; // nincs elég ház

            int wood = home.Stock.GetValueOrDefault(Resource.Wood, 0);
            int stone = home.Stock.GetValueOrDefault(Resource.Stone, 0);
```

```
            bool canStoneBuild = w.StoneBuildingsEnabled &&
home.CanBuildWithStone && stone >= home.HouseStoneCost;
            bool canWoodBuild = wood >= home.HouseWoodCost;

            // Utility heurisztika
            float uBuild = needHousing ? 0.8f : 0.2f; // ha kell ház, erős
motiváció
            if (canStoneBuild || canWoodBuild) uBuild += 0.15f;

            float desiredWood = home.HouseWoodCost * 2; // célkészlet
(heurisztika)
            float uWood = Math.Clamp(1f - wood / Math.Max(1f, desiredWood),
0f, 1f) * 0.7f;

            float uIdle = 0.05f; // kis alapzaj, hogy néha pihenjen

            // kicsi sztochasztika, hogy ne legyen teljesen egysíkú
            uBuild += (float)_rng.NextDouble() * 0.05f;
            uWood  += (float)_rng.NextDouble() * 0.05f;

            if (uBuild >= uWood && uBuild >= uIdle) return GoalId.BuildHouse;
            if (uWood >= uBuild && uWood >= uIdle) return GoalId.GatherWood;
            return GoalId.Idle;
        }

        public Job DecideJob(Person p, World w)
            => DecideGoal(p, w) switch
            {
                GoalId.GatherWood => Job.GatherWood,
                GoalId.BuildHouse => Job.BuildHouse,
                _ => Job.Idle
            };

        // Célirányos lépegetés a legközelebbi erőforrás felé (egyszerű, V0)
        public void StepTowardsNearestResource(Person p, World w, Resource
res)
        {
            // 1) Megkeressük a legközelebbi res tile-t (naiv O(W*H))
            (int x, int y)? best = null; int bestDist = int.MaxValue;
            for (int x = 0; x < w.Width; x++)
            for (int y = 0; y < w.Height; y++)
            {
                var t = w.GetTile(x, y);
                if (t.Type != res || t.Amount <= 0) continue;
                int dx = x - p.Pos.x, dy = y - p.Pos.y;
                int d = Math.Abs(dx) + Math.Abs(dy); // manhattan-táv
                if (d < bestDist) { bestDist = d; best = (x, y); }
            }

            // 2) Ha nincs, kóborlás kis zajjal
            if (best == null)
```

```
            {
                WanderLike(p, w);
                return;
            }

            // 3) Egy lépés a cél felé (rácslépés, akadálykezelés később)
            int sx = Math.Sign(best.Value.x - p.Pos.x);
            int sy = Math.Sign(best.Value.y - p.Pos.y);
            var nx = Math.Clamp(p.Pos.x + sx, 0, w.Width - 1);
            var ny = Math.Clamp(p.Pos.y + sy, 0, w.Height - 1);
            p.Pos = (nx, ny);
        }

        public void WanderLike(Person p, World w)
        {
            // A meglévő Wander mintájára, de itt külön modulban
            int step = Math.Max(1, (int)p.Home.MovementSpeedMultiplier);
            p.Pos = (
                Math.Clamp(p.Pos.x + _rng.Next(-step, step + 1), 0, w.Width
- 1),
                Math.Clamp(p.Pos.y + _rng.Next(-step, step + 1), 0, w.Height
- 1)
            );
        }
    }
}
```

2) `Person.cs` – minimális módosítások

A fájl tetején add hozzá:

```
using WorldSim.Simulation.AI; // ÚJ
```

A `Person` osztályban (mezők között) add hozzá:

```
private readonly AgentBrain _brain = new(); // ÚJ: Utility-alapú döntéshozó
```

A `switch (Current)` blokkban cseréld az `Idle` ágat és a mozgáshívásokat:

```
case Job.Idle:
    // RÉGI: véletlen választás GatherWood vs BuildHouse
    // ÚJ:
    Current = _brain.DecideJob(this, w);
    break;

case Job.GatherWood:
    if (w.TryHarvest(Pos, Resource.Wood, 1))
        _home.Stock[Resource.Wood] += w.WoodYield;
```

```
    else
        _brain.StepTowardsNearestResource(this, w, Resource.Wood); // ÚJ:
célirányos lépés
    // opcionális: néha térjünk vissza Idle-be, hogy újraértékeljen
    if (_rng.NextDouble() < 0.02) Current = Job.Idle;
    break;

case Job.BuildHouse:
    int colonyPop = w._people.Count(p => p.Home == _home);
    int capacity = _home.HouseCount * w.HouseCapacity;
    if (colonyPop >= capacity)
    {
        if (w.StoneBuildingsEnabled && _home.CanBuildWithStone &&
_home.Stock[Resource.Stone] >= _home.HouseStoneCost)
        {
            _home.Stock[Resource.Stone] -= _home.HouseStoneCost;
            _home.HouseCount++;
            w.AddHouse(_home, Pos);
            Current = Job.Idle;
        }
        else if (_home.Stock[Resource.Wood] >= _home.HouseWoodCost)
        {
            _home.Stock[Resource.Wood] -= _home.HouseWoodCost;
            _home.HouseCount++;
            w.AddHouse(_home, Pos);
            Current = Job.Idle;
        }
        else
        {
            // nincs elég nyersanyag → gyűjtés felé indulunk
            _brain.StepTowardsNearestResource(this, w, Resource.Wood);
            if (_rng.NextDouble() < 0.02) Current = Job.Idle; // időnként
újraértékel
        }
    }
    else
    {
        // nincs sürgős housing-igény → ne erőltesse
        Current = Job.Idle;
    }
    break;
```

> **Megjegyzés:** a fenti módosítások minimálisra fogják a diffet, és nem nyúlnak a reprodukció/halálozás logikájához. Ha szeretnéd, a következő iterációban szétszedjük `Person.Update`-ot „életszimuláció” + „AI döntés” részekre.

3) (Opcionális) Debug overlay – „miért ezt tetted?”

A következő körben érdemes egy egyszerű telemetry-t felrajzolni: az utolsó `GoalId` és 0..1 utility értékek kirajzolása a kolónia HUD-ján. (Ha kéred, adok kész rajzoló metódust `SpriteBatch`-hez.)

Következő lépés (V1): GOAP + HFSM + Fuzzy

- **GOAP**: `MoveTo(resource) → Harvest → Deposit → BuildHouse` akciógráf (A* a worldstate-en).
- **HFSM**: `Idle/Walk/Run/Turn` állapotok a `MoveTo` alatt; `desiredSpeed`-et egyszerű fuzzy adja (távolság/Threat/Energy).
- **BT**: a GOAP által tervezett akciók reaktív végrehajtása (abort, fallback).
- **Data-driven**: célok/akciók JSON-ban.

Ha rábólintasz, a V1-hez feltöltöm a `Simulation/AI/` alá a GOAP/BT/HFSM fájlokat és a szükséges hívásokat a `World.Update` / `Person` oldalán. - **v0 (1–2 nap)**: Needs + Utility selector + pár BT akció (Eat/Sleep/Work/MoveTo). - **v1 (1 hét)**: GOAP 5–10 akcióval (AcquireFood pipeline), cooldown/inercia, telemetria overlay. - **v2**: HTN rutinok (napi rend), social steering, influence map. - **v3**: LLM-hook (opcionális), dialógus-eszközök.

Gyors tesztötletek

- „Éhség 0.9, nincs kaja, van pénz, bolt 200m” → Terv: MoveTo(Bolt)→Buy→MoveTo(Otthon)→Cook→Eat.
- „Energia 0.2, munka 30 percen belül” → Alvás röviden **ha** belefér (Utility kombináció idő-súlyozással).
- Két ügynök keskeny folyosón: steering + szabály: *yield to right*.

Tipp a debughoz

- **BT/GOAP vizualizáció**: aktuális fa csomópont-állapot színezve, GOAP open/closed list méret.
- **Heatmap**: influence map a népsűrűségre/veszélyre.
- **Event log**: "why did you do X?" – listázd a top-3 célt pontszámmal és a kiválasztott tervvel.

Integráció a másik modell javaslataival (HFSM, Fuzzy, ECS)

Az ajánlások **teljesen kompatibilisek** a jelen vázunkkal. A javasolt „best-of” hibrid:

- **Utility AI**: célprioritás (Needs/Traits/Context → utility).
- **GOAP/HTN**: több-lépéses terv (cél → akciósor).
- **Behavior Tree**: reaktív végrehajtás/megszakítás.
- **HFSM (hierarchikus állapotgép)**: alacsony szintű **mozgás/animáció** kontroll (Idle/Walk/Run/Turn/Stop), finom átmenetekkel és „substate”-ekkel.
- **Fuzzy moduláció**: folytonos paraméterek (sebesség, kíváncsiság, kockázatvállalás) számítása, amely **Utility-t** és **HFSM átmeneteket** is befolyásol.
- **ECS + EventBus + Data-driven**: komponens-alap, események, JSON konfiguráció.

HFSM – Locomotion/Animation réteg

```
[Locomotion]
  ├─ Idle
  ├─ Move
  │   ├─ Walk
  │   └─ Run
  └─ Turn
```

Átmenetek (példa): `Idle→Move` ha `desiredSpeed>ε`; `Walk↔Run` ha fuzzy `runIntent` küszöböt átlép; `Move→Turn` ha heading error > küszöb; minden állapotból `→Stop` vészben.

BT integráció: a BT „MoveTo(target)” levéli **csak** a célpozíciót és sebességcél-t (desiredSpeed) adja; a **HFSM** intézi a konkrét átmenetet és animációt.

GOAP integráció: a „MoveTo(X)” akció effektusa a világállapotban, de a **fizikai mozgást** a HFSM végzi.

Fuzzy moduláció (folytonos döntések)

- **Bemenetek**: távolság a célig, akadály-sűrűség, fáradtság (Energy), veszély (Threat), Personality (kockázat, lelkesedés).
- **Tags** (háromszög/trapéz): *közeli/közép/távoli*; *alacsony/közepes/magas*; stb.
- **Szabályok** (példa):
- HA `távolság` = távoli ÉS `Threat` = alacsony → `runIntent` = közepes
- HA `távolság` = közeli VAGY `akadály` = magas → `runIntent` = alacsony
- **Kimenet**: `desiredSpeed ∈ [walkSpeed, runSpeed]`, ami **HFSM** váltásokat triggerel + **Utility** finomhangolást ad (pl. sietség növeli „ArriveOnTime” cél utility-jét).

Minimális C# interfész-váz HFSM-hez

```
public interface IState { void Enter(); void Exit(); void Tick(float dt); }
public sealed class HFSM
{
    private IState _current;
    public void Set(IState s){ _current?.Exit(); _current = s;
_current.Enter(); }
    public void Tick(float dt){ _current?.Tick(dt); }
}
// Példa: WalkState/RunState az Agent Movement komponensét vezérli (speed/
animation).
```

ECS térkép (részlet)

- **Components**: `Transform`, `Movement`, `LocomotionHFSM`, `Needs`, `Traits`, `Perception`, `PlannerState`, `BehaviorTreeRef`.
- **Systems**: `PerceptionSystem`, `UtilitySystem`, `PlanningSystem`, `BTSystem`, `LocomotionSystem`, `PathfindingSystem`, `SteeringSystem`.
- **Events**: `GoalChanged`, `PlanBuilt`, `ObstacleDetected`, `ChatIntent`, `Arrived`, `Stuck`.

Eseményvezérelt interfész (chatbot-ready)

- `ChatIntent(name:"go_to", args:{target:"bolt"})` → **Utility** megnöveli a kapcsolódó cél súlyát.
- `executePlan("nyisd ki az ajtót")` → egy **Intent→Goal** leképező modul GOAP-célra fordítja.
- Visszacsatolás: `Why did you do X?` → utolsó **Utility** táblázat + GOAP terv + aktív BT csomópont.

Mikor melyik dominál?

- **HFSM**: *„hogyan mozogjak most?”* (állapotok/animációk, milliszekundumos döntések)
- **BT**: *„melyik alfeladatot futtassam most?”* (reaktivitás)
- **GOAP/HTN**: *„milyen akciósor vezet a célhoz?”* (másodperces/terv)
- **Utility**: *„melyik célt üldözzem?”* (motivációválasztás)

Zárás

Ez a váz stabil, jól skálázódó alapot ad intelligensebb, mégis determinisztikusan debuggolható ügynökökhöz. A későbbi LLM integrációt az Intent/Tool réteg készíti elő, anélkül, hogy a szimulációs konzisztenciát feladnád.