Townlet – Authoritative Design Reference

Version: 1.0
Date: 2025-09-29
Status: RECOVERED FROM DOCUMENTATION

Executive Summary

Townlet is a small-town life simulation aimed at demonstrating emergent behaviors in a population of reinforcement-learning agents [1]. The project's proof-of-concept targets **6–10 agents** inhabiting a 48×48 grid world with basic needs (hunger, hygiene, energy) and daily routines. The system integrates a **deterministic tick-based simulator**, an RL training harness, and an operator UI. Key objectives include achieving stable agent life-cycles, implementing conflict dynamics (queues and rivalry), and integrating an RL policy loop without destabilizing the simulation. As of this recovery, Townlet's design documents are in place and many core systems are implemented, but some planned features remain unfinished or need alignment. This reference design consolidates the intended architecture, requirements, and decisions to guide the project's completion.

Project Architecture

System Overview

Townlet is structured as a **headless simulation core** with modular subsystems, orchestrated by a central tick loop. Surrounding the core are an RL training interface and an observer UI/console for human interaction [2] [3]. The simulation runs in real-time ticks (~250 ms each) and exposes both a programmatic API (for training and automation) and a console/telemetry feed (for monitoring and ops) [4] [5]. External components such as a PettingZoo RL environment wrapper and (in the future) an optional 3D renderer or pathfinding microservice can integrate via defined interfaces [6] [7].

High-Level Component Diagram: (Console/API and Telemetry feed into the simulation loop; simulation loop integrates World, Queue/Lifecycle, Reward/Stability, etc., and outputs to UI)

```
┌──────────────┐    ┌──────────────┐    ┌──────────────┐
│  Console/API │    │   Telemetry  │    │  Observer UI │
└──────┬───────┘    └──────┬───────┘    └──────┬───────┘
       │                  │                   │
       ▼                  ▼                   │
┌───────────────────────────────────────────────────────┐
│              Core Simulation Process                 │
│  ┌──────────────┐  ┌──────────────┐  ┌───────────┐  │
│  │ World Model  │  │ Queue & Life │  │ Reward &  │  │
│  │  + Config    │  │ cycle Mgmt   │  │ Stability │  │
│  └──────┬───────┘  └──────┬───────┘  └─────┬─────┘  │
│         │                 │                │    │
│  ┌──────▼───────┐ ┌───────▼──────┐  ┌─────▼────┐ │
│   │ Observation  │   │ Policy Runner │    │ Snapshots  │ │
│   │ Builder      │   │  (RL Env)     │    │ & Config   │ │
│   └──────────────┘   └───────────────┘    └────────────┘ │
└──────────────────────────────────────────────────────────┘
```

Control Flow: Each simulation tick proceeds in a fixed sequence [5] : 1. **Console Commands Applied** – Pending admin commands (spawn, toggle features, etc.) are validated and applied to the world state (if any) [8] .
2. **Perturbations Triggered** – Random or scheduled world events (e.g. price spikes, outages) are evaluated (currently planned but stubbed) [9] [10] .
3. **Policy Decision** – Every agent receives an observation, and the Policy Runner (either scripted behavior or RL model) chooses an action for each agent [11] .
4. **Affordance Resolution** – The World Model processes agent actions: queueing for interactive objects, granting reservations, executing affordances (e.g. using an item), etc. The Queue Manager arbitrates access and may enforce fairness delays or ghost steps to break deadlocks [12] [13] . Completed affordances apply effects (needs changes, item usage) to agents and emit events [14] [15] .
5. **Needs & Lifecycle Updates** – Agents' needs decay and are checked against exit conditions. The Lifecycle Manager flags agents for termination if conditions are met (e.g. hunger collapse or unemployment criteria) [16] [17] , queues replacements, and logs exit events [18] [19] .
6. **Reward Calculation** – The Reward Engine computes each agent's reward for the tick based on homeostatic needs and other factors (capped to configured limits) [20] [21] . Guardrails (clipping, no positive reward on death) ensure training stability [22] [23] .
7. **Telemetry Publication** – A structured snapshot of the tick's outcome is published: observations, rewards, key events, and metrics are packaged by the Telemetry Publisher [24] [25] . The system records any notable alerts (e.g. threshold breaches) via the Stability Monitor at this stage [26] [27] .
8. **RL Step & UI Update** – In training mode, the Policy Runner (PPO harness) logs the transition (state, action, reward) for learning [28] [29] . In parallel, the Observer UI (console dashboard) refreshes its display by pulling the latest telemetry snapshot [30] [31] .

This loop repeats indefinitely or until a configured tick limit.

Core Components

Below are the major components of Townlet, with their design intent and status:

- **Core Simulation Loop** (`townlet.core.sim_loop.SimulationLoop`):
- **Purpose:** Central orchestrator that ties all subsystems together for each tick [32] . It enforces the tick timeline and invokes the world, policy, lifecycle, reward, telemetry, and stability modules in order [5] .
- **Responsibilities:** Maintain the master clock (`tick`), call each subsystem in sequence, and aggregate per-tick artifacts (observations, rewards) for logs/tests [33] [26] . Also handles resetting the simulation to an initial state as needed [34] .
- **Interfaces:** Provides `step()` and `run()` methods to advance the simulation [33] . Exposes tick outputs via a `TickArtifacts` data class for external consumption or testing [35] [5] .
- **Dependencies:** Constructs all subsystems on init (WorldState, PerturbationScheduler, ObservationBuilder, PolicyRuntime, RewardEngine, TelemetryPublisher, StabilityMonitor) [36] . It relies on each to implement their contract each tick.

- **Acceptance Criteria:** The tick loop should process events in the correct order, never deadlock, and respect real-time pacing (~4 ticks/sec). A smoke test running 1000 ticks should complete without error and basic metrics (e.g. no starving agents in M0 scenario) should meet expectations [37]. Current design indicates the loop meets ordering requirements [38] and passes long-run smoke tests (2+ days) with no regressions [39] [40].

- **World Model** (`townlet.world.grid.WorldState` and related classes):

- **Purpose:** Represents the game world state including the grid, agents, objects, and affordances. This is the authoritative simulation state that evolves each tick [41] [42].
- **Responsibilities:** Manage agent locations and needs, track interactive objects and their queues, execute affordances, and handle persistent world data (economy, weather, etc.). It encapsulates logic for things like feeding, working, and other agent interactions with the environment [43] [44]. Also enforces world constraints (e.g. one agent per tile reservation, inventory stock changes).
- **Interfaces:** Provides methods to register objects/affordances (`register_object`, `register_affordance`) for setup [45] [46]. Key methods include `apply_actions(actions)` to apply agent actions to the world [47] [48], `resolve_affordances()` to progress any ongoing affordances and trigger events [49] [50], and `snapshot()` to get a copy of agent states [51]. It also exposes specialized queries for local views and diagnostics (e.g. `local_view()` for observation, `active_reservations`, and event draining) [52] [53].
- **Data:** Defined sub-structures include `AgentSnapshot` (agent state: position, needs, wallet, job info, etc.) [54] [55], `InteractiveObject` (objects like fridge, stove, bed with occupancy and stock) [56], and `AffordanceSpec`/`RunningAffordance` for actions that take time [57] [58]. The world holds mappings of agents, objects, and running affordances. It also maintains *rivalry ledgers* per agent to quantify conflict over time [59] [60].
- **Dependencies:** Uses `QueueManager` for queueing logic, `EmbeddingAllocator` for ID slot assignment, and the simulation config (which includes definitions for jobs, affordances YAML, etc.) [61] [62]. Hooks to `townlet.policy` and `telemetry` to produce events and metrics.

- **Acceptance Criteria:** The world should correctly apply action effects and generate expected events/metrics. For example, using an affordance should deduct resources and add a completion event, queueing should enforce one agent active per object with others waiting [63] [14]. Post-tick world state must reflect need decay, any location changes, and updated rivalry stats. Documented invariants like “no stuck reservations after ghost step” or “needs clamped [0,1]” must hold [63] [64]. (Note: Some features like weather or utilities are noted in design but not yet implemented – see Unresolved Design Questions.)

- **Queue & Reservation Manager** (`townlet.world.queue_manager.QueueManager`):

- **Purpose:** Handle multi-agent access to interactive objects through queues, ensuring fairness and preventing deadlock in contested resources [63] [65].
- **Responsibilities:** Maintain per-object FIFO queues and track which agent (if any) currently holds a reservation. Applies fairness policies: after an agent uses an object, others get priority while the prior user faces a cooldown, and queue starvation is mitigated by slight priority to longer-waiting agents [66] [67]. It also counts consecutive failed access attempts and triggers a one-tick “ghost step” to break deadlocks when an agent fails to progress after N tries [68] [69].
- **Interfaces:** Key methods are `request_access(object_id, agent_id)` – returns True if agent can immediately use the object or False if queued [70] [71]; `release(object_id, agent_id, success)` – frees a reservation (on affordance completion or giving up) and, if success, starts the agent’s cooldown timer [72] [73]; and `on_tick()` – to expire any cooldowns each tick [74] [75]. It also offers `queue_snapshot()` for debugging and `metrics()` to expose counts of fairness events (cooldowns applied, ghost steps triggered) [76] [77].

- **Dependencies:** Called by WorldState when agents attempt "request" actions or complete affordances [48] [44]. Relies on config for parameters like `cooldown_ticks` and `ghost_step_after` (stall threshold) [78] [79].

- **Acceptance Criteria:** Queues should behave as specified in Requirements [80]: no agent can bypass the queue unfairly (cooldown entries enforce that repeat access is delayed [81] [82]), and deadlocks are resolved by ghost steps (after a configurable number of blocked attempts, the blocking agent is forced to move, logged as a ghost step event [69]). Unit tests or simulations should confirm that when two agents continuously conflict over a resource, ghost steps occur and eventually each gets access [65] [83]. The metrics counters (`cooldown_events`, `ghost_step_events`) should increase appropriately during such scenarios [84] [85]. The component meets these criteria (per design and initial testing) and is **fully implemented**.

- **Perturbation Scheduler** (`townlet.scheduler.perturbations.PerturbationScheduler`):

- **Purpose:** Introduce random city-wide events or environmental perturbations (price surges, power outages, special meetings) on a schedule or probabilistically [10] [86]. This simulates external pressures and ensures agents face varied scenarios (especially in later milestones).
- **Responsibilities:** Manage timers and triggers for events defined in the config (with daily rate limits, cooldown buckets, etc. [87]). When triggered, apply effects to the World (e.g., suppress exits during an outage) and emit event logs. It should coordinate with Lifecycle to enforce grace periods around events (e.g. no evictions during a city event plus some ticks) [87] [88].
- **Interfaces:** Method `tick(world, current_tick)` called each tick to possibly enqueue or execute events [9]. Also an `enqueue(events)` method to accept externally scheduled events [89]. In design, events would propagate to world state (e.g., via hooks or direct WorldState methods).
- **Dependencies:** Uses configuration for event definitions and rates [87]. Would interact with WorldState to apply effects (notably with Lifecycle for exit suppression and Telemetry to log events).

- **Acceptance Criteria:** Should fire events at the configured frequency (e.g., ensure probability of >2 overlapping events <5% per day [87]). Each event type's effects (e.g., price_spike doubling prices, an arranged_meet forcing two agents together) should be traceable in world state and telemetry. As of now, **this component is stubbed** – the `tick()` method contains a TODO and does not yet implement any events [9]. Completing this is critical for Milestone M4 onward, but the absence does not affect earlier milestones beyond lacking some variability.

- **Lifecycle Manager** (`townlet.lifecycle.manager.LifecycleManager`):

- **Purpose:** Enforce agent life-cycle rules such as agent termination (exits) and spawning of new agents. It encapsulates conditions for failure (e.g. starvation, eviction, unemployment) and ensures the population refreshes in a controlled way [90] [91].
- **Responsibilities:** Each tick, evaluate each agent for termination criteria and output who should be removed (`terminated[agent_id]=True/False`) [17] [16]. Specifically:
    - **Basic Needs Exit:** If an agent's critical need (like hunger) falls to zero, mark them to exit (simulating death or collapse) [16].
    - **Unemployment/Exit Cap:** If employment enforcement is on, apply the **exit-cap policy** for missed shifts. Agents with too many absences enter a pending exit queue; the manager processes this queue each day, capping immediate exits to `daily_exit_cap` and deferring others with events for ops review [92] [93]. Manual approvals (`employment_exit approve`) or deferrals from ops are also handled here [94] [92].

    - (Future) **Other Exits:** Design calls for eviction (losing home), chronic neglect, or age-out conditions, and these would be checked here, but they are not yet implemented.
    - It also resets daily counters at the appropriate interval (e.g., resets `exits_today` each new day) [95].
- **Interfaces:** The primary interface is `evaluate(world, tick) -> dict[agent_id, bool]` which returns termination flags [17]. Internally it has helpers like `_evaluate_employment()` for the unemployment logic [96] [97]. It does *not* directly remove agents; it signals termination, and presumably the simulation or an external process would handle the actual removal and potential respawn (the spawning mechanism is currently rudimentary or manual).
- **Dependencies:** Accesses the `WorldState` for agent stats (needs, job status) and for invoking world methods to queue or execute exits [92] [18]. Uses config thresholds (daily_exit_cap, max_absent_shifts, etc. from `employment` settings) [98] [99]. Works with TelemetryPublisher to emit events like `employment_exit_processed` when an exit occurs [18] [19].

- **Acceptance Criteria:** No agent should persist in a failed state beyond the allowed limits: e.g., hunger <=0 triggers an exit immediately [16], and an agent with `>=3` absences in 7 days should be queued for exit (if daily cap is reached, they remain pending with an event) [100] [99]. The exit-cap policy must prevent more than N exits in one day to avoid population collapse [100] [98]. The system achieves the intended **hybrid approach** (Option C in design) – individual threshold plus global cap – as confirmed by the decision matrix [100] [101]. The code implements this policy (absent agents go to a queue, daily processing with cap, etc.) and passes internal tests for queue behavior [93] [102]. However, other life-cycle rules (eviction, aging) are **not yet implemented**, and spawn/replacement logic is minimal. These are noted as future work.

- **Observation Builder** (`townlet.observations.builder.ObservationBuilder`):

- **Purpose:** Construct the structured observation (state representation) for each agent each tick. Townlet supports multiple observation *variants* (full, hybrid, compact) with different fidelity, but the **hybrid** variant is the primary focus [103] [104]. The observation includes an egocentric map around the agent and a vector of features describing the agent's state and environment [105] [106].
- **Responsibilities:** For each agent, gather local grid information and agent-specific stats into a consistent tensor. The hybrid variant outputs:
    - An `11×11` local grid map (centered on the agent) with multiple channels (self, other agents, objects, reservations) [107] [106]. *(Currently, object and reservation channels are placeholders until world geometry is implemented)*.
    - A feature vector (~531 dimensions) covering agent needs, status flags, time-of-day encoding, employment metrics (attendance ratio, shift state one-hot, wages withheld), and conflict metrics (max rivalry and count of rivals above threshold) [106] [108] [109]. It also reserves 16 slots for future social features (currently zeros) [110] [111].
    - Metadata including the observation variant, map shape, channel labels, and feature names for reference [112] [113].
- **Interfaces:** The main method `build_batch(world, terminated)` returns a dict of `agent_id -> {map, features, metadata}` for all agents [114] [115]. It handles marking a `ctx_reset_flag` in the feature vector if an agent was terminated this tick (signaling an episode boundary to the policy) [116] [115]. Internally, `_build_single(world, snapshot, slot)` does the heavy lifting for one agent [117]. The builder uses a stable `EmbeddingAllocator` to assign each agent a persistent ID slot number and includes that normalized slot in the features [118].
- **Dependencies:** Relies on `WorldState.local_view()` to get nearby entities (currently uses agent positions; object positions are not yet spatially indexed, hence placeholders) [119] [120]. Uses `world.rivalry_top(agent, limit)` to retrieve top rivalry values [109]. The

`ObservationVariant` config dictates which variant to build; hybrid is default and fully implemented, while other variants fall back to dummy outputs [121].

- **Acceptance Criteria:** The output tensors must align with the spec [107] and be consistent between tick-to-tick unless underlying state changes. The hybrid observation should include all the fields enumerated in the specification [105] [108]. A sample hybrid observation has been validated and published in docs (e.g., `observation_hybrid_sample.npz`) [122] [123]. Unit tests confirm shapes and presence of key values (needs, time encodings, rivalry metrics) [124] [125]. This component is **completed** for the hybrid mode (with extension points for future variants). Future enhancements will populate the currently blank map channels for objects once world coordinates are tracked, and incorporate social data once relationships are implemented (see Unresolved Questions).

- **Embedding Allocator** (`townlet.observations.embedding.EmbeddingAllocator`):

- **Purpose:** Provide each agent with a stable **embedding slot ID** for observation and neural network consistency. When agents die and are removed, their slot is eventually recycled after a cooldown to avoid overlapping identities [126] [127]. This enables fixed-size observation spaces even as agents come and go.
- **Responsibilities:** Maintain a pool of slot IDs (max slots configured, e.g., 64). When a new agent appears, assign the lowest available slot; if all slots are in use or cooling down, it will **force-reuse** the oldest released slot (with a warning) [127] [128]. It tracks when slots were released to enforce a cooldown period (default 2000 ticks ~ 2 days) before reuse [127] [129]. Also accumulates metrics like how often forced reuse happens and the current reuse rate.
- **Interfaces:** Methods `allocate(agent_id, tick)` returns the slot number for the agent (assigning a new one if first time) [130] [131], `release(agent_id, tick)` frees an agent's slot at removal time [132]. `metrics()` provides telemetry such as total allocations, forced reuse count, and a `reuse_warning` boolean if the forced reuse rate exceeds a threshold (meaning identity collisions might be high) [133] [134].
- **Dependencies:** Used by ObservationBuilder to allocate slots each tick before building features [116] [115]. Integrated into TelemetryPublisher to expose embedding reuse metrics and warnings for the UI/ops [135] [136]. Configured via `embedding_allocator` settings (max_slots, cooldown_ticks, reuse_warning_threshold).

- **Acceptance Criteria:** No two living agents share the same embedding slot at any time (unless intentionally forced when capacity exceeded). After an agent exits, its slot is not reused until either the cooldown passes or the system runs out of free slots [127] [129]. The `forced_reuse_rate` should generally stay low; if it exceeds the threshold (e.g., >10%), the system raises a warning for ops [135] [136]. The implemented allocator meets these criteria, and tests ensure that releasing and reallocating slots behave as expected (including the warning flag) [137] [138].

- **Policy Runner (RL Environment)** (`townlet.policy.runner.PolicyRuntime` and `TrainingHarness`):

- **Purpose:** Interface between the simulation and decision-making logic. In Townlet, this serves both as the PettingZoo-like environment wrapper for RL algorithms and as a host for scripted behaviors. It collects observations, applies an RL policy or scripted policy to choose actions, and accumulates trajectories for learning [139] [140]. The Policy Runner is designed to handle multi-agent decisions and integrate with a PPO training loop.

- **Responsibilities:**
    - **Action Selection:** For each agent, call either a **BehaviorController** (scripted rules) or an RL policy network to determine the next action. Currently, a placeholder scripted policy returns basic intents (idle/wait or simple moves) because learning is not fully live yet [141] [142]. The runner packages each chosen action into a uniform format (with fields for action kind, target object, etc.) and assigns each a unique `action_id` for logging [143] [144].
    - **Trajectory Recording:** After each step, record the rewards and done flags for each agent to their trajectory entries [28] [145]. When observations for the new tick are available, finalize the transition frames (state, action, reward, next state) and store them in a trajectory buffer [29] [146]. This is used for training updates.
    - **Policy Outputs:** If a neural policy network is present, annotate each trajectory frame with the policy's log-probability and value estimate for the action (to support advantage estimation). The runner lazily initializes a `ConflictAwarePolicyNetwork` (a simple neural net) when needed and computes `log_prob` and `value_pred` for each action using the network [147] [148]. If no torch is available or network not initialized, it defaults these to 0, effectively treating it as a random policy stub [149].
    - **RL Integration Hooks:** The Policy Runner works with the **TrainingHarness** for offline training. It flushes trajectories to the harness and can run special modes (replay, rollout capture) to generate training data. (See **RL Training Harness** below for details.)
- **Interfaces:** `decide(world, tick)` – returns a dict of agent actions for the tick [11] [141]. `post_step(rewards, terminated)` – store rewards and done signals after the world update [28]. `flush_transitions(observations)` – combine the latest observations with stored actions/rewards into completed trajectory frames [29] [146]. The runner also exposes internal methods for policy network management (`_ensure_policy_network`) and uses a static configuration for the telemetry schema version (currently `1.1` for training logs) [150].
- **Dependencies:** Uses `townlet.policy.behavior` module (scripted policy rules) and optionally `townlet.policy.models` (neural network, PyTorch). The SimulationLoop calls `policy.decide()` then later `policy.flush_transitions()` each tick [151]. The TelemetryPublisher uses policy outputs indirectly via the training logs. Config flags like `features.stages` or `training.curiosity` could alter behavior selection (e.g., enabling learned behaviors in later phases), though at present the behavior is mostly hardcoded.

- **Acceptance Criteria:** In pure scripted mode, agents should at least perform no invalid actions and respond to basic needs (the current placeholder behavior likely just idles or random-walks). When integrated with an RL model, the environment (observations, rewards, dones) must be correctly provided each step to ensure learning stability [140] [152]. The PettingZoo ParallelEnv interface expectations (like consistent agent indexing and termination signals) are conceptually met by our structure (we maintain a consistent `action_id` mapping and mark done agents) [143] [149]. The design also requires an **option commitment** window of 15 ticks for high-level options [139] [153]; this is not explicitly implemented yet in code (scripted behavior doesn't enforce it), so that remains a minor gap. Overall, the Policy Runner is functioning in a minimal capacity but will evolve as learned policies come online.

- **Reward Engine** (`townlet.rewards.engine.RewardEngine`):

- **Purpose:** Compute each agent's reward signal for RL training, capturing the multi-objective nature of the simulation (survival, productivity, social fulfillment, etc.) while applying safety guardrails to stabilize learning [154] [155].

- **Responsibilities:** Calculate a single scalar reward per agent per tick by combining various factors:
    - Base survival reward (a small positive value each tick to encourage staying alive, e.g., `survival_tick` weight) [156] [157].
    - Need satisfaction penalties: for each need (hunger, hygiene, energy), subtract a penalty proportional to its deficit squared (encouraging agents to keep needs high) [21]. The weights for each need penalty are configurable [158].
    - Additional shaping rewards/punishments per design, such as punctuality bonus for arriving at work on time, wage earnings, social interaction bonuses, or penalties for eviction/death, etc. [20] [159]. (Many of these are described in docs but not all are implemented yet.)
    - Apply clipping: limit the reward to ± `clip_per_tick` to avoid extreme values [20] [160]. If an agent is terminated (`done`), also cap reward at 0 or negative to avoid giving a dying agent a positive boost [161] [162].
- **Interfaces:** A single method `compute(world, terminated) -> Dict[agent_id, float]` runs each tick after lifecycle evaluation [163] [164]. It fetches each agent's current needs from the world and computes the reward as described. The output is a dictionary of reward values for each agent. There is no internal state kept between ticks.
- **Dependencies:** Uses config `rewards` section for weights and clipping values [156]. Needs data and other agent info come from `WorldState` and `AgentSnapshot`. In the future, it might consider world events or social context (design mentions chat rewards and conflict avoidance bonuses in later phases [165] [166]) – those would require data from Telemetry or world on relevant events. Currently none of those extras are included in code.

- **Acceptance Criteria:** Agents who meet basic survival goals (keeping needs >0) over time should accumulate positive reward, whereas agents who neglect needs or get terminated should accumulate less or negative reward. The reward function should be tuned such that, for example, one missed meal's penalty outweighs a small punctuality bonus (as per design guidance [167]). In implementation, the present reward calculation covers needs and survival clipping, meeting the fundamental requirement of discouraging need deficits and preventing reward explosion via clipping [21]. However, more nuanced terms (punctuality, social rewards) and long-horizon normalization (like running mean for reward normalization [168]) are **not yet implemented**. These will need to be added as the corresponding features (employment punctuality, social interactions) come online.

- **Stability Monitor & Promotion Manager** (`townlet.stability.monitor.StabilityMonitor`):

- **Purpose:** Track key performance indicators (KPIs) over time and decide when an RL policy is stable enough to **promote** to production (or when to rollback due to regressions). Also serves as a central alarm system for undesirable conditions (e.g., starvation spikes, thrashing behaviors) by raising alerts if thresholds are crossed [169] [170]. Essentially, it implements the "canary" tests and gating rules for releasing learned policies.
- **Responsibilities:** On each tick, ingest the latest telemetry stats and update alert status:
    - Monitor counts of critical events: lateness spikes (if cumulative late ticks for the day exceed threshold) [171], starvation (e.g., any starvation event would trigger), option thrashing (rapid switching of behaviors), and other guardrails defined in config [172] [173].
    - Check embedding reuse warnings (from EmbeddingAllocator) and queue conflict metrics for anomalies [174] [175].
    - Aggregate metrics over rolling windows (24h of sim time ~ 1000 ticks as per design) [170]. If the system holds two consecutive stable windows without alerts, it would mark a policy

    as ready for promotion (design mentions needing two good windows for promotion) [170]. Conversely, if certain canaries go off (e.g., lateness_spike alert), it could flag for rollback.
  - Provide the latest alert status for ops dashboards or automated triggers (e.g., storing `latest_alert` if any) [176] [174].
- **Interfaces:** `track(tick, rewards, terminated, queue_metrics, embedding_metrics, job_snapshot, events, employment_metrics)` is called each tick after telemetry publishing [26] [27]. It analyzes the provided data: for example, it counts `affordance_fail` events and compares against `affordance_fail_threshold` config to raise an alert [177], or sums lateness from the job snapshot to check against `lateness_threshold` [178]. It sets an internal `latest_alert` string if any condition triggers (like `"lateness_spike"` or `"employment_exit_backlog"`) [179]. It also caches the last seen metrics for queue and embedding for potential use (like drift detection).
- **Dependencies:** Relies on TelemetryPublisher outputs (queue metrics, embedding metrics, events, etc.). Uses config `stability` thresholds for various canaries [180] [181]. It's conceptually linked to an external promotion controller (not implemented) that would query it or get notified when conditions are met to flip a policy from "shadow" to "release" and vice versa [182] [183].

- **Acceptance Criteria:** The StabilityMonitor should detect when something is going wrong (e.g., if after enabling a new feature, starvation events start occurring, it should set an alert). Specifically, in employment loop tests, if pending exits exceed the queue limit, it should raise `employment_exit_queue_overflow` [179]; any pending exits at all (but below limit) raises a milder `employment_exit_backlog` [179]. These conditions are implemented. Other planned canaries like option thrash (too frequent policy switching) or sustained reward regression would be coded similarly but are not yet present (marked as TODO in code) [27]. For promotion gating, the rule is two clean evaluation windows – since we lack the full multi-run evaluation harness in code, this is tracked procedurally via CI and ops for now. The component is **partially implemented**: basic alerts work (embedding reuse, affordance failures, lateness, exit queue) [174] [179], but more sophisticated monitoring and automatic promotion logic remain to be built.

- **Telemetry & UI Gateway** (`townlet.telemetry.publisher.TelemetryPublisher` and `townlet_ui.telemetry.TelemetryClient`):

- **Purpose:** Bridge the simulation's internal state to external visibility. The TelemetryPublisher gathers each tick's data into snapshots and events for consumption by UIs or logging systems [184] [25]. The observer UI (a Rich Text console dashboard) uses a TelemetryClient/Console bridge to retrieve this data in near-real-time to display simulation metrics and agent status [185] [31].
- **Responsibilities:**
  - The **TelemetryPublisher** collects critical data at the end of each tick: queue stats, rivalry snapshots, embedding metrics, latest events, per-agent job snapshots, economy snapshot, and employment metrics [25] [186] [187]. It buffers console commands from the operator to apply next tick [188]. It also notifies any subscribers (like a UI event stream) of new events each tick [189] [190]. (Currently, it does not send data over a network – it's in-memory, as the UI runs in-process via the console).
  - The **TelemetryClient/ConsoleRouter** is an abstraction for the UI to query telemetry. A ConsoleRouter registers commands like `telemetry_snapshot` (to get the full snapshot), `employment_status`, etc., which the UI can call as if they were console commands [191] [192]. The TelemetryClient in the UI simply invokes these and gets structured data (internally via TelemetryBridge that reads from TelemetryPublisher) [193] [194]. This design allows the UI to use the same interface as a human operator would via console commands.

    - The Observer UI Dashboard (Rich CLI) then renders the telemetry snapshot: it shows schema version and warnings, an employment queue panel (pending exits, count, cap) [195] [196], a conflict panel (queue cooldown and ghost step counters, number of rivalry links) [197], an agent table with each agent's wallet, shift state, attendance, etc. [198] [199], and an ASCII map (with S for self, A for agents) of the local view for a selected agent [200] [201]. It updates continuously every second by advancing the sim loop and pulling a fresh snapshot [202].
- **Interfaces:** TelemetryPublisher's primary interface is `publish_tick(tick, world, observations, rewards, events)` called by the SimLoop [24] [184]. Console-facing interfaces include `queue_console_command(cmd)` and `drain_console_buffer()` for operator input injection [188]. The ConsoleRouter (in `townlet.console.handlers`) maps string commands to handlers that return telemetry data or perform world actions [191] [203]. Notably, `employment_exit` commands call into world methods to approve or defer an exit [203] [204].
- **Dependencies:** Depends on WorldState for all data (agents, objects, events) each tick and on subsystems like QueueManager and EmbeddingAllocator for metrics. The UI tools depend on the `rich` library for rendering. Config's `telemetry.schema_version` is managed here (currently "0.3.0") [205] [206].

- **Acceptance Criteria:** The telemetry system should output a consistent snapshot each tick that accurately reflects the simulation state. Key fields in the snapshot (like each agent's shift_state, attendance_ratio, exit_pending flag) must match the WorldState's data [186] [187]. In testing, the team captured sample telemetry JSON and ensured the UI can parse and display it [207] [208]. The UI should update in near real-time and highlight important conditions (e.g., turns the employment panel red if any exit backlog exists [209], conflict panel color if ghost steps occurred [210]). Current implementation meets these: the Rich dashboard shows live employment/backlog status and conflict metrics as intended, based on our dry-run tests [211] [212]. Future enhancements include an optional web-based UI after milestone M2.5, but the console UI is considered sufficient for MVP [213] [214]. (Note: TelemetryPublisher's network/pub-sub output is stubbed with a TODO [215] – it does not yet push data to an external service, which is acceptable for now.)

- **Console & Admin Commands** (`townlet.console.handlers.ConsoleRouter` and Typer CLI scripts):

- **Purpose:** Allow developers and operators to interact with the running simulation: spawning agents, adjusting needs, reviewing or overriding exits, etc., and to run various modes (e.g., start training, run a rollout capture). The console system also handles authentication modes (viewer vs admin) for a future multi-user environment [216].
- **Responsibilities:** Provide a set of **commands** that can be issued at runtime. In design, planned commands include:
    - `spawn <agent>` or scenario loading (to add agents),
    - `setneed <agent> <need> <value>` to force-adjust needs,
    - `teleport <agent> <x> <y>` to reposition an agent,
    - `promote_policy <checkpoint>` and `rollback_policy <snapshot>` to manage policy versions,
    - `employment_status` and `employment_exit <review|approve|defer>` to manage the unemployment queue,
    - etc. [30] [217].

    In the current implementation, the **ConsoleRouter** has handlers for a subset of these: telemetry snapshot and employment-related commands are implemented and integrated with the UI/ops flow [218] [203]. For example, `employment_exit review` returns the list

    of pending exits so ops can decide, and `employment_exit approve <id>` triggers that agent's removal (via WorldState) and logs an event [203] [204]. The `telemetry_snapshot` command returns all current telemetry data for external tools or logging [191] [218].
    - Other commands are scaffolded: e.g., the design mentions console controlling feature flags, but toggling flags is currently done via config or code injection, not a typed command. The CLI scripts (`scripts/run_simulation.py`, `scripts/run_training.py`) use Typer to expose some functionality (like running the sim with a config, or running a training harness in various modes). The console authentication (viewer vs admin) is noted in docs [219] but since there is no multi-user network service yet, it's not enforced in code except conceptually.
- **Interfaces:** The ConsoleRouter/handlers provide programmatic command dispatch within the app (used by TelemetryClient and potentially by tests). Separately, **Typer CLI** entry points exist: running `python scripts/run_simulation.py` will start a simulation loop (with optional tick-limit) and possibly expose a basic CLI (though currently the rich UI replaces that). `scripts/run_training.py` supports modes `--mode replay|rollout|mixed` to run training harness workflows (like capturing rollouts or running PPO on a manifest) [220] [221]. These scripts parse command-line arguments and call into the `TrainingHarness` accordingly.
- **Dependencies:** Hooks deeply into TelemetryPublisher and WorldState for data and mutating actions. The console commands for employment use world methods like `employment_request_manual_exit` we saw earlier [204]. Future commands like `spawn` would call a world method to create a new agent (not currently present; in tests they seed via scenarios instead).

- **Acceptance Criteria:** Operators should be able to retrieve key status info and perform limited control actions without stopping the simulation. The implemented subset (telemetry snapshot, employment queue management) meets the immediate needs of demonstrating conflict intro: e.g., approving an exit via console immediately removes the agent from pending and the UI reflects it [203] [204]. Security-wise, sensitive commands (spawning, promotions) should require admin mode, but since everything runs locally now, this is low priority. The main gap is that many intended commands are **not yet implemented**; these are documented but not functional, meaning full interactive control is limited at this stage.

- **RL Training Harness** (`townlet.policy.runner.TrainingHarness` and associated Replay/PPO modules):

- **Purpose:** Coordinate end-to-end training of reinforcement learning models using Townlet simulation data. It supports offline training on recorded rollouts (replay), live rollout generation, and the PPO optimization loop. Essentially, it's the component that ties simulation trajectories to a learning algorithm (Proximal Policy Optimization) and produces updated policies.
- **Responsibilities:** Provide different run modes for training:
    - **Replay Loading:** Load saved observation/reward sequences (from NPZ/JSON samples) to verify they contain conflict metrics and compute basic stats [222] [223]. This helps test that the observation tensor and telemetry can be read by training code. The harness implements `run_replay(sample_path)` to load one sample and print conflict-aware stats (like rivalry feature means) [224]. `run_replay_dataset(manifest)` goes through a list of samples and can aggregate stats [225] [226]. These fulfill Work Package goals of demonstrating that rivalry signals flow into training data [227] [228].
    - **Rollout Capture:** Run the simulation for N ticks and collect all trajectory frames into a `RolloutBuffer`. The harness's `capture_rollout(ticks, ...)` uses an internal SimulationLoop to generate data, optionally seeding default agents if none present [229]

[230]. It saves the captured data to disk (JSONL logs and manifest) for offline analysis [231] [232]. This was used to produce the scenario logs for Phase 4 acceptance testing [233] [234].
    - **PPO Training Loop:** Take a dataset (either from a live capture or a configured ReplayDataset from files) and run a PPO optimization for a given number of epochs. The harness's `run_ppo()` builds or uses an in-memory dataset of transitions, sets up a simple neural network (if using torch), and iterates through the data to compute losses and adjust the policy [235] [236]. It logs typical PPO metrics like policy loss, value loss, entropy, KL divergence, etc., and enforces gradient clipping [237] [238]. The harness updates a telemetry log (NDJSON) with per-epoch summary fields including conflict statistics (e.g., rivalry feature averages) [150] [239]. In practice, the current implementation uses a stub model and dummy loss calculations (the actual optimization is present but the model is simplistic). It ensures compatibility with schema version 1.1 for these logs [240].
    - **Outputs & Monitoring:** The harness can output training logs to a file and periodically print summaries. It respects a `max_log_entries` to rollover log files to avoid huge single files [239]. The team integrated these outputs with CI – the logs are saved as artifacts and the ops handbook describes analyzing them [241] [242].
- **Interfaces:** Methods include `run_replay`, `run_replay_batch`, `run_replay_dataset` for offline data; `capture_rollout` and `run_rollout_ppo` to generate and immediately train on fresh data [243] [232]; and `run_ppo` for the core training loop on a given dataset [235]. The harness is invoked by CLI (e.g., `scripts/run_training.py --mode mixed` will use these to do a rollout + train cycle).
- **Dependencies:** Requires the SimulationLoop (to simulate rollouts), the ReplayDataset utilities (which handle batching and conflict feature validation), and PyTorch for the PPO network and optimizer. Config files provide hyperparameters (learning rate, batch sizes in `ppo` config, etc.). Telemetry from simulation is consumed via the RolloutBuffer (which records events like queue_conflict counts per cycle) [244].
- **Acceptance Criteria:** The training harness succeeded in executing Phases 2–4 of PPO integration [245] [246], as evidenced by the completion certificate [247] [248]. Criteria included:
    - Conflict features appear in training batches (met, rivalry_max and avoid_count are included and non-zero in logs) [239] [241].
    - The system can run a combined **mixed mode** (alternating learning from replay and live rollout) without crashing, and log the results [244] [249].
    - The telemetry logs for PPO contain the new fields (cycle_id, conflict stats, etc.) with correct values, and the validation script passes on them [250] [251].
    - CI automation was able to run a short mixed scenario and produce artifacts as expected [252] [253].

      By those measures, the harness is **operational in a basic form**. However, it uses a dummy policy network (random initialization) and does not yet implement training continuation or advanced PPO features like advantage normalization beyond a single run. For MVP and integration testing, it's sufficient; for real learning progress, more work is needed (e.g., saving/loading models, curriculum over milestones, etc.). These extensions are planned in later phases.

Data Architecture

Townlet's data model can be viewed in three layers: **Agent State**, **World State**, and **Telemetry/External State**.

- **Agent State Model:** Each agent is represented by an `AgentSnapshot` which holds:
- **Identity:** `agent_id` (a unique string, also used as key in dictionaries) and an **embedding slot** (managed by the allocator) for stable representation [254] [118].

- **Position:** grid coordinates `(x, y)` representing location in the town [255].
- **Core Needs:** a dict of need levels (hunger, hygiene, energy) normalized 0.0–1.0 [256] [257]. These decay over time and are replenished by certain affordances (e.g., eating raises hunger).
- **Economic Status:** `wallet` (currency on hand) and possibly inventory items (e.g., groceries, though currently wages accumulate as an inventory item "wages_earned" for tracking) [258] [259].
- **Employment Status:** `job_id` (if employed), and fields for tracking attendance: `on_shift` flag, `shift_state` (pre_shift, on_time, late, absent, etc.), `lateness_counter` (current shift lateness in ticks), `late_ticks_today`, `attendance_ratio` (recent shifts attendance fraction), `absent_shifts_7d`, `wages_withheld` (total pay lost due to absences), and `exit_pending` (if queued for removal) [260] [261] [55]. These are updated by the Employment Loop logic as agents attend or miss work.

- **Relationships:** Currently only conflict/rivalry metrics are tracked. Each agent has a rivalry ledger mapping other agents to a "conflict tension" score [59] [60]. This grows when they conflict in queues and decays over time. Social friendship metrics are planned but not implemented yet (placeholders exist in observation output but always zero) [111].
  These agent fields flow into observations (for the agent's own view) and telemetry snapshots (for operators to monitor individuals). Data integrity: the design ensures no negative needs (clamped at 0), and values like attendance_ratio stay between 0 and 1 [262] [263].

- **World State & Environment Model:** The world is a grid of fixed size (48×48 by default) with various **interactive objects** placed (fridge, stove, bed, workplace, etc.) [43]. Rather than storing a full grid matrix in memory, the world primarily stores objects and agents with their coordinates, and computes local views on the fly. Key data structures:

- **Objects Registry:** `WorldState.objects` is a dict of `object_id -> InteractiveObject`. Each InteractiveObject has a type and possibly an `occupied_by` field and a `stock` (inventory of that object) [264]. For example, a fridge might have stock of "meals" count, a stove might have "raw_ingredients", etc., to simulate resource usage [45]. Affordances reference object types, e.g., "cook_meal" applies to a stove and will consume ingredients from that stove's stock.
- **Affordance Registry:** `WorldState.affordances` holds all affordance definitions (`AffordanceSpec`) loaded from YAML config at startup [265] [46]. Each affordance spec includes duration (ticks it takes) and effects (changes to agent needs or world on completion). These specs drive the creation of RunningAffordances when an agent starts one.
- **Reservations & Queues:** `WorldState._active_reservations` tracks which agent currently holds each object (if any). `_running_affordances` tracks ongoing affordances by object (since an object can be in use for the duration) [266] [44]. The QueueManager, outside of WorldState, holds waiting lists for each object, but WorldState provides methods like `_sync_reservation()` to keep its active reservation map in sync after queue updates [69] [267].
- **Pending Events:** WorldState accumulates gameplay events in `_pending_events` (grouped by tick) and provides `drain_events()` each tick for telemetry to consume [53]. Events can include `affordance_start/finish/fail` with details, `queue_conflict` events when ghost steps or handovers occur, and `employment_exit_pending/processed` events from lifecycle [15] [19]. This event log is the basis for telemetry event streams and testing invariant conditions.
- **Economy & Other Systems:** Some global fields like `store_stock` (aggregated inventory of all stores/objects) and economic parameters (prices, wages) come from config and are used in affordance effects but not heavily dynamic in code yet [268] [269]. Weather or utilities are acknowledged in design but not present in data structures yet (future extension).

- **Configuration & Snapshots:** `SimulationConfig` contains sub-configs (like feature flags, queue_fairness, employment settings, reward weights, etc. as seen in docs/REQUIREMENTS.md) [270] [156]. A copy of this config is held in WorldState for reference. The simulation supports saving a snapshot of the entire world for pause/resume, which would include all agents and objects, RNG state, and the config identity [271]. This is not fully exposed in UI yet, but the design ensures that a snapshot .pkl contains everything needed to resume the sim in identical state [271].
  Data Flow: Each tick, data flows from WorldState → ObservationBuilder (to produce observations) → Policy → actions → back to WorldState (state mutation) → TelemetryPublisher (snapshots out). Also, console input flows into WorldState (via TelemetryPublisher's buffer and `apply_console` method).

- **External Data & Integration Points:**

- **Telemetry Outputs:** Townlet defines a versioned telemetry schema for the observer interface [150]. At schema v0.3.0, the telemetry JSON includes sections for jobs, economy, conflict, events, etc. [194]. A changelog (docs/TELEMETRY_CHANGELOG.md) is maintained as fields are added – for instance, adding rivalry fields bumped schema from 0.2.0 to 0.3.0 [272] [273]. External tools or UIs are expected to validate the `schema_version` and possibly show a warning if there's a mismatch (the console does this via `_schema_metadata`) [274].
- **Checkpoints and Policy Data:** RL policy models (if trained) would be saved as checkpoint files. The design envisioned console commands to promote or rollback policies using these files [275]. Currently, the harness does not produce a trained policy file (since training is stub), so this integration is deferred. In future, `promote_policy <file>` would instruct the live simulation to swap in a new policy network for the Policy Runner, and `rollback_policy` would revert to a prior snapshot [276] [277]. The architecture leaves room for this by separating PolicyRuntime (which could load different networks).
- **Integration with PettingZoo/AI Libraries:** The Policy Runner is conceptually aligned with PettingZoo's interface (each tick corresponds to an environment step with parallel agents) [278]. For now, training happens via our custom harness, but one could wrap SimulationLoop and PolicyRuntime into a PettingZoo ParallelEnv if needed. Dependencies like stable-baselines3 or RLlib are not directly integrated but could consume the observation/reward stream if we expose it. The system's modular design (especially the separation of simulation from training harness) supports plugging in such libraries in the future.

Integration Points

Townlet is largely self-contained, but it touches or could touch a few external systems and libraries:

- **Reinforcement Learning Libraries:** The design chooses **PettingZoo** as the API standard for multi-agent RL environment [279] [140]. The simulation provides observations and reward in a format that is compatible with PettingZoo environments (e.g., a dictionary of agent observations each step, with a mechanism to mark agents done) [28] [29]. Internally, Townlet's RL loop uses a custom PPO implementation. We use **PyTorch** for neural network and optimization (optional, enabled if installed) [240] [280]. If PyTorch is absent, the code safely continues with dummy outputs, ensuring the sim can run without GPU/torch (useful for pure simulation mode) [149]. Going forward, integration with frameworks like stable-baselines3 could be explored to leverage their algorithms; our data structures (ReplayDataset, RolloutBuffer) can be adapted for that.
- **Rich/Textual UI Library:** The Townlet observer UI relies on the **Rich** library for terminal graphics. We have integrated Rich to display tables and colored panels for agent stats and system metrics [195] [281]. We also considered alternatives (Streamlit, custom web UI) and decided to stick with CLI for now due to simplicity and low dependency overhead [282] [283]. This means the

UI runs in-process and requires no separate service – a design decision to expedite development and maintain flexibility for a future graphical client. The Rich console approach has been validated in a dry-run demo with positive feedback, and a potential **FastAPI+React** thin client is noted as a stretch goal post-MVP if needed [284].

- **Configuration and Persistence:** Townlet uses **YAML configuration files** for scenario setup (e.g., `configs/examples/*.yml` for base simulation parameters, lists of affordances, etc.). The system has a config loader that uses Pydantic for validation and to ensure all required keys are present [285] [286]. Snapshots of the simulation can be saved to disk (pickled WorldState with its agents and objects). These snapshots along with logs can be archived for analysis or debugging. For instance, Phase 4 training outputs are stored under an `artifacts/phase4/` directory in the repo, including logged summaries of runs [287] [244].
- **Potential External Service (Pathfinding):** The high-level design notes an optional Rust/C++ pathfinding microservice that could be plugged in if we needed more advanced or faster navigation than our grid lookups [288]. This is not implemented yet – currently, movement is trivial (agents move on grid with no complex path planning needed). The architecture foresees adding it behind a stable C ABI if needed, toggled by a feature flag, without altering the rest of the sim [288] [7].
- **Issue Tracking and CI:** While not an integration "service," it's worth noting that Townlet's development process uses GitHub for issue tracking and CI. The CI pipeline runs tests and even executes a short simulation + training cycle to ensure nothing breaks performance or basic metrics [252] [253]. This environment ensures that integration points (like the training harness with torch) are continuously validated.

Non-Functional Requirements

Townlet's design accounts for several non-functional goals:

- **Performance:** The simulator should handle at least **8 agents at 4,000 ticks/sec** on CPU-only hardware in full observation mode [289] [290]. This is a stretch goal; current implementation runs ~0.3 sec per tick with the Rich UI on (approx 3.2 ms/tick measured) [291], which is slower due to debug overhead but acceptable for development. Without UI, tick performance is closer to target (the tick benchmark showed <5μs overhead per tick with job loop features enabled, which is negligible) [292]. The plan to meet scale is to optimize hot paths (affordances, neighborhood queries, reward calc) and consider using **Numba/Cython** for Python speedups or offload to native code for pathfinding as needed [289] [293]. So far, profiling shows tick time increases <5% even after adding employment logic (verified by micro-benchmarks) [292], staying within the ±5% regression budget set in risk R7 [294] [295]. We also collect performance metrics in QueueManager (nanosecond timing for each operation) to identify bottlenecks [296] [297].
- **Scalability:** The architecture is built to scale to dozens of agents by design. Key measures: the spatial queries are O(n) per agent by default, but design suggests adding a spatial index if agents grow to 50+ (e.g., quadtree for neighbor lookups) [63] [298]. The rivalry and queue systems are per pair or per object, which scale roughly O(n^2) in worst case (all agents conflict with all), but typical use will see localized conflicts. The modular design allows running multiple simulation shards in parallel for training scale-out: e.g., one could run many headless sim instances to generate experiences concurrently (we haven't implemented multi-process coordination yet, but it's feasible given the state encapsulation). Memory scalability is aided by using numeric tensors for observations and only storing needed state (no huge 2D arrays for the whole grid, we compute local views on the fly).
- **Security:** As a simulation tool not exposed as a public service, security is mainly about safe handling of commands and data integrity. We have **token-gated console modes** planned (viewer vs admin) [219] and an audit log of console commands (the console router can log every

command with issuer and result) [216] . Privacy mode is another aspect: design says telemetry can hash agent IDs for public sharing [299] . We have a stub for `privacy_mode` where if enabled, TelemetryPublisher would output hashed IDs instead of real ones [299] . Compliance like GDPR or streamer mode would rely on this. Since these aren't critical in our dev environment, they're noted but not fully implemented. On a basic level, we ensure no crash or undefined behavior from malformed commands: unrecognized console commands are caught and reported safely [300] . The code has a SECURITY.md policy (reporting process for vulnerabilities) [301] – although Townlet is pre-release, we aim to address any security issues promptly.

- **Maintainability:** Maintainability is emphasized through clear module boundaries and thorough documentation. Every subsystem corresponds to a document and a test suite (e.g., Observation spec doc + tests, Employment design briefs + tests, etc.), ensuring developers can trace why something is built a certain way [302] [303] . The repository follows standard Python style (enforced via linting and type checks as described in AGENTS.md guidelines) [304] . There is a strict PR checklist that includes updating documentation and tests for any feature affecting them [305] [306] – for example, risk R6 was addressed by adding a docs/tests checklist that must be ticked off before merge [307] [308] . Additionally, versioning of docs (with semantic tags like v1.0.0 on sign-off) and an issue tracking workflow ensure changes are tracked [309] . The design deliberately separated concerns (simulation vs UI vs training) which helps different team members work without stepping on each other (e.g., RL engineers can tweak TrainingHarness without touching core sim code, and vice versa).
- **Reliability & Robustness:** We have incorporated several guardrails to keep the simulation stable over long runs: telemetry backpressure handling (it will drop or aggregate diffs if the UI can't keep up, to avoid stalling the sim) [310] , console idempotency tokens (commands can include a `cmd_id` to ensure a repeated command isn't applied twice) [311] , and careful handling of edge cases like all embedding slots in use (raises a clear error or warning) [312] . Long-running smoke tests (≥2 days simulation) are part of our verification, and the system must survive those without memory leaks or cumulative errors [40] . So far, deterministic fixed-seed runs are used to catch any divergence. The chaos testing plan (random teleports, toggling outages, etc.) is outlined in Requirements [313] [314] and will be applied as features mature, but is partially done informally (no major crashes observed under the tests we've run). We log incidents and will use the risk register to track any reliability issues discovered.
- **Extensibility:** Though not explicitly listed, it's worth noting as a quality: the system is built to allow new features to "slot in" via interfaces. For example, adding a new need or a new type of affordance mostly involves updating config and perhaps ObservationBuilder, without rewriting the core loop. The documentation plan includes an **Architecture & Interfaces Guide** to serve as the authoritative reference for module responsibilities [315] – effectively this document itself – which ensures that future contributors can understand where to add new logic. The feature-flag system (stages A/B/C and on/off toggles) allows incomplete features to be present but disabled in production runs, aiding gradual integration [316] [317] .

Implementation Standards

The project follows standard Python development practices and some project-specific conventions:

- **Coding Conventions:** Code uses 4-space indentation and is (largely) PEP8 compliant. Naming is consistent: modules and functions in `snake_case`, classes in `PascalCase`, constants in `UPPER_SNAKE_CASE` [304] . We leverage Python 3.11's typing features extensively – functions and methods are type-annotated, and `from __future__ import annotations` is used to allow forward references [318] [319] . Data classes (`@dataclass`) are used for simple state containers (AgentSnapshot, QueueEntry, etc.) to reduce boilerplate [254] [320] . Inline comments mark future work with a consistent tag `# TODO(@townlet): ...` which is easily greppable [321] . There is an existing pattern of placing internal helper methods with an underscore prefix within classes, grouping public API at the top and details at the bottom (e.g., QueueManager's `_assign_next`) [322] [323].

- **Testing Requirements:** All new features must have corresponding tests. The repository is organized with `tests/` mirroring the `src/` layout [324] [325]. For example, `test_observer_ui_dashboard.py` covers the UI rendering, `test_policy_models.py` covers the neural network shapes, `test_employment_loop.py` was created to validate attendance and exit logic [326]. We aim for high coverage on critical modules (core sim, queue, lifecycle). Some tests use property-based approaches for invariants (like ensuring no deadlocks in extended queue fuzzing) as planned in the test plan [327] [328]. There are markers for long-running tests (e.g., soak tests could be marked `slow` to exclude them in quick runs) [329] [330]. Continuous Integration runs all tests on each commit, and also performs a static analysis (ruff and mypy) to catch style and type issues [331] [332]. The acceptance criteria include specific test scenarios – for example, "long-running smoke (≥2 days) with metrics diff to baseline" as a verification item [40] – which will be automated as the project stabilizes (likely as part of a nightly test suite).

- **Documentation Standards:** We maintain a comprehensive `docs/` directory that covers program management (roadmaps, work packages) and technical design (requirements, design decisions, architecture). Each major feature has either a brief or spec document – e.g., conflict intro, observation tensor spec, attendance design brief – and these were kept up to date through the recent development sprints [125] [263]. The **Documentation Plan** explicitly enumerated all needed documents (Product Charter, Architecture Guide, Ops Handbook, etc.) and their owners [333] [334]. We have delivered most of these in draft form, except the Product Charter (vision doc) which was implicitly covered by conceptual design, and a formal Data & Privacy Policy which remains to be written (Week 5 in doc plan) [335] [336]. Code is also documented with docstrings in critical places (e.g., every public class and method in the core has a docstring explaining usage and behavior [337] [32]). Moreover, we keep an **Implementation Notes** log (changelog) to record technical decisions, cross-linked with docs updates and date stamps [338] [339]. This helps new contributors to quickly understand recent modifications and their rationale.

- **Security Practices:** Although not a networked app, we treat any external input (configs, console commands) with care. YAML config loading is done via a safe parser and validated by Pydantic models to avoid injection of arbitrary code or invalid values causing undefined behavior [340]. The console interface, when extended for network use, will enforce auth tokens (design suggests a bearer token and viewer vs admin roles) [219]. Logging of console actions (with `cmd_id`) ensures traceability and prevents replay attacks if we ever run Townlet as a service [311]. We also have guidelines not to commit any sensitive info and to use environment variables for any secrets (though currently none are needed) [341]. When the project moves towards a possible cloud or multiplayer scenario, we'll need a thorough security review, but for the single-user local scenario the measures above are sufficient.

Unresolved Design Questions

While the recovered design is largely coherent, a few gaps and ambiguities remain where further clarification or decisions are needed:

- **Social/Relationship System:** Milestone M3 ("Relationships Observed") is upcoming [342], but the design for how friendships and social interactions influence behavior is not yet detailed in the documentation. We have placeholders in observations (social snippet vectors) [343] [110] and references to chat rewards and narration phases [344] [166], but no concrete spec for how agents form or use relationships. We will need a design brief for the social system (similar to the employment briefs) to define trust/friendship mechanics and how they feed into options and rewards. Until then, these aspects remain disabled (feature flag `relationships: OFF`).

- **Advanced Console Commands:** The design lists many console commands (spawn, setneed, etc.), but currently only telemetry and employment commands are implemented. It's unclear how some commands should behave precisely – e.g., `spawn` could create an agent with default state or require a template; `force_chat <agent1> <agent2>` (mentioned in docs as a console command) would require the social system. We should decide which admin interventions are highest priority and define their effects. For now, we note that **operational control is limited** – if an agent gets stuck other than via employment exit, we don't have commands to directly fix it (aside from possibly writing a one-off script). This is acceptable for development but needs addressing for a production scenario or a live demo with human operators.

- **Pathfinding and World Geometry:** As noted, the current movement model is simplistic (agents move directly on the grid with Manhattan distance). The architecture allows plugging in a pathfinding module [288], and there's an outstanding question of when to introduce that. If the environment grows in complexity (e.g., obstacles requiring A* pathfinding), we must either implement a Python version or integrate an external library. The decision on technology (Rust vs Python) and timing is pending. It's flagged as not needed for Phase A–C unless performance dictates (since the grid is open and small). We will re-evaluate if movement behavior becomes a bottleneck or if richer navigation (like finding a path around other agents or buildings) is desired in Phase C or later.

- **Reward Function Tuning:** The current reward shaping is basic; design documents mention more nuanced tuning (like ensuring a "skipped meal" penalty outweighs any punctuality reward [167], or adjusting curiosity bonus by milestone [345]). We haven't explicitly parameterized those in config beyond what's in REQUIREMENTS.md defaults. As we approach Phase C (when learned policies fully replace scripted), we may need to revisit the reward weights. There's no immediate conflict in docs about this, just an open question of calibration: are the default weights in REQUIREMENTS (e.g., hunger weight 1.0, hygiene 0.6, etc. [156]) producing the desired agent behavior? This likely requires empirical tuning once agents learn, and might involve updating the design to reflect findings. It's understood that some reward adjustments ("guardrails") might be done dynamically (e.g., value normalization across phase flips [168]) – we've left hooks in code for running mean tracking, but not fully utilized them.

- **Policy Promotion Workflow:** The concept of release vs shadow policy and promotion gates is described in design [346] [347] and ops docs, but practically, we have only one policy active (since learning isn't continuously running yet). The `promote_policy` console command is noted as "TBD implementation" in the Ops Handbook [348], meaning we haven't decided how exactly to switch the active policy on the fly. Will it load a saved network into the running simulation? Or will we always stop and restart with a new policy? This needs a decision. Given the modular PolicyRuntime, a runtime swap is possible but requires careful state handover. For now, this remains unresolved – we assume manually stopping the simulation and restarting with a new policy checkpoint for major promotions (which is acceptable in an experimental phase, if not elegant). As we move to a more automated pipeline, we'll need to implement this command or an equivalent mechanism.

- **Data & Privacy Policy Document:** As per the documentation plan, a formal data/privacy policy was slated for Week 5 [349]. It hasn't been produced yet. While not affecting the software's behavior, this is important for open-sourcing or for using Townlet in public demonstrations (ensuring observers understand what data is logged, how long it's kept, etc.). The system already supports basic privacy mode toggling (hashing IDs) [299], but questions like log retention, personal data (if any), and compliance need answering. We should draft this document, especially as we approach a stable release, to align with legal and community best practices.

- **Pending Design Approvals:** Some design decisions were made pending stakeholder approval (as noted in work package docs). For example, the exact values for daily exit cap (default 2) and absence penalties were approved by product and architecture leads [350], but ops input was "N/A (not staffed)" [351]. If an ops team member later joins or if we simulate operator feedback, these might be revisited. Another example: the Observer UI's decision to stick with CLI was to be confirmed post-usability test [352]. We have done an initial dry run, but should formally capture that feedback and confirm if any UI changes are needed now (the **usability review is effectively ongoing**). These are not conflicts per se, but areas to double-check with stakeholders to ensure the recovered design aligns with expectations.

In conclusion, the Townlet design as assembled here represents the "true intent" of the project – a stable core simulation with modular extensions for conflict and social dynamics, geared toward evaluating RL agents in a complex environment. The next steps are to address the few incomplete features and refine implementations per this design reference. The accompanying Implementation Status Report will detail the current state of each component and what remains to be done to fully realize this design.

---

1 README.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/README.md

2 3 4 7 42 64 135 137 140 150 219 239 240 278 279 347 ARCHITECTURE_INTERFACES.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/ARCHITECTURE_INTERFACES.md

5 8 11 24 26 32 33 34 35 36 38 151 163 sim_loop.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/core/sim_loop.py

6 12 41 43 63 90 126 139 154 169 216 288 HIGH_LEVEL_DESIGN.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/HIGH_LEVEL_DESIGN.md

9 89 318 337 perturbations.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/scheduler/perturbations.py

10 37 86 165 327 342 344 MILESTONE_ROADMAP.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/MILESTONE_ROADMAP.md

13 14 15 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 69 83 119 254 255 258 264 265 266 267 268 269 321 grid.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/world/grid.py

16 17 18 19 92 93 94 95 96 97 102 manager.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/lifecycle/manager.py

20 22 65 80 87 88 91 103 104 127 152 153 155 156 159 161 166 167 168 170 172 256 270 271 289 290 293 298 299 310 311 313 314 316 317 328 343 345 346 REQUIREMENTS.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/REQUIREMENTS.md

21 23 157 158 160 162 164 engine.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/rewards/engine.py

25 77 184 186 187 188 189 190 205 206 215 259 319 publisher.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/telemetry/publisher.py

27 171 173 174 175 176 177 178 179 180 181 monitor.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/stability/monitor.py

28 29 141 142 143 144 145 146 147 148 149 224 225 226 229 230 231 232 235 236 237 238 243 280 runner.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/policy/runner.py

30 185 207 208 211 217 220 221 275 276 277 348 OPS_HANDBOOK.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/OPS_HANDBOOK.md

31 195 196 197 198 199 200 201 202 209 210 281 dashboard.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet_ui/dashboard.py

39 40 WORK_PACKAGE_CONFLICT_INTRO.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/WORK_PACKAGE_CONFLICT_INTRO.md

66 67 68 70 71 72 73 74 75 76 78 79 81 82 84 85 296 297 320 322 323 queue_manager.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/world/queue_manager.py

98 99 100 101 350 351 EMPLOYMENT_EXIT_CAP_MATRIX.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/design/EMPLOYMENT_EXIT_CAP_MATRIX.md

105 106 107 110 112 120 122 123 OBSERVATION_TENSOR_SPEC.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/design/OBSERVATION_TENSOR_SPEC.md

108 109 111 113 114 115 116 117 118 121 257 builder.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/observations/builder.py

124 125 WORK_PACKAGE_OBSERVATION_UPGRADE.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/WORK_PACKAGE_OBSERVATION_UPGRADE.md

128 129 130 131 132 133 134 136 138 312 embedding.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/observations/embedding.py

182 183 245 246 MASTER_PLAN_PROGRESS.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/MASTER_PLAN_PROGRESS.md

191 192 193 194 203 204 218 274 300 handlers.py
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/src/townlet/console/handlers.py

212 WORK_PACKAGE_OBSERVER_UI.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/WORK_PACKAGE_OBSERVER_UI.md

213 214 282 283 284 352 OBSERVER_UI_DECISIONS.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/design/OBSERVER_UI_DECISIONS.md

222 223 227 228 WORK_PACKAGE_CONFLICT_TRAINING.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/WORK_PACKAGE_CONFLICT_TRAINING.md

233 241 242 247 248 250 COMPLETION_CERTIFICATE_PPO_PHASE2_4.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/COMPLETION_CERTIFICATE_PPO_PHASE2_4.md

234 244 249 251 252 253 287 ROLLOUT_PHASE4_ACCEPTANCE.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/ROLLOUT_PHASE4_ACCEPTANCE.md

260 261 262 263 EMPLOYMENT_ATTENDANCE_BRIEF.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/design/EMPLOYMENT_ATTENDANCE_BRIEF.md

272 273 291 292 308 326 338 339 340 IMPLEMENTATION_NOTES.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/IMPLEMENTATION_NOTES.md

285 286 302 309 315 333 334 335 336 349 DOCUMENTATION_PLAN.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/DOCUMENTATION_PLAN.md

294 295 303 305 306 307 WORK_PACKAGE_EMPLOYMENT_LOOP.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/docs/WORK_PACKAGE_EMPLOYMENT_LOOP.md

301 SECURITY.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/SECURITY.md

304 324 325 329 330 331 332 341 AGENTS.md
https://github.com/tachyon-beep/townlet/blob/c027861cdc7cf448a5eb23b17eeaad2e83f0e88e/AGENTS.md