ChatGPT

# Observations Submodule (`townlet.observations`)

The **Observations** submodule generates structured observation data for each agent at every simulation tick. Its core class, `ObservationBuilder`, **constructs per-agent observation payloads** using the current world state and simulation config [1]. The observation payload includes a **feature vector** of agent and environment state, an optional **local map tensor** (grid around the agent), and associated **metadata** like feature names and map dimensions [2]. The `ObservationBuilder` is initialized with the simulation's observation settings (e.g. `ObservationVariant` type and parameters from config) and pre-computes indices for various features and social slots.

- **Feature Encoding:** The builder compiles a rich feature vector covering agent needs (hunger, hygiene, energy), status (wallet money, lateness, on-shift flag), time of day (sin/cos), attendance/wages stats, last action outcome, shift state (one-hot across "pre_shift", "on_time", etc.), and contextual flags [3] [4]. It normalizes the agent's embedding slot index and includes an episode progress fraction [5]. Environmental flags are added (e.g. whether the agent currently has an active reservation or is waiting in an object's queue) [6] [7]. If pathfinding hints are enabled, features for the direction to the nearest target landmark (like a stove) are computed [8] [9]. These features are all inserted into a NumPy array with a fixed index mapping.

- **Local Map:** For observation variants that include spatial context (e.g. "hybrid" or "full"), the builder produces a **local occupancy map** centered on the agent. It queries the world for a local view around the agent and fills a tensor with channels marking the agent's own position, other agents, objects, and reservation sites in nearby tiles [10]. Additional channels can encode normalized direction vectors toward the agent for each occupied tile [11]. The map size (window) comes from the config. In **"compact" mode**, the builder omits the map and uses an empty placeholder array [12], relying solely on the feature vector.

- **Social Features:** The builder optionally appends a **"social snippet"** vector representing relationships. It allocates slots for top friends and rivals per agent based on config (embedding each other agent's ID via a hash and including trust/familiarity/rivalry metrics) [13] [14]. Summary statistics like mean/max trust and rivalry can also be included [15]. If the full relationship system is not enabled, the code falls back to using the rivalry ledger – for example, if friendship data is unavailable it populates rival slots with the highest rivalry scores and zero trust [16] [17]. *(This is a noted gap: the comment "fallback to rivalry ledger until full relationship system lands" indicates the relationship feature is provisional pending a more complete social system)* [16]. The social snippet logic can be disabled entirely if relationships are turned off in the config, in which case `relation_source` is marked as "disabled" and no social data is added [18].

- **Observation Assembly:** For each agent each tick, `ObservationBuilder.build_batch` produces an observation dict: it takes a **snapshot** of the world state for all agents and then for each agent, allocates an embedding slot and builds the observation [19]. It calls an internal `_build_single` based on the configured variant to assemble the map, features, and metadata [20] [21]. The metadata includes the observation variant name, map shape, channel names, and the list of feature names (with indices) [2]. In all variants the result is a dictionary with keys `"map"`, `"features"`, and `"metadata"` for each agent [22]. Before returning, the builder also sets a **context reset flag** in the features if the world or policy signaled a context reset or if the agent was terminated this tick (to inform the agent that its internal memory/context should reset) [23]. Overall, the Observations submodule centralizes how raw simulation state is encoded into model-ready tensors, making it a critical interface between the environment and policy.

# Rewards Submodule (`townlet.rewards`)

The **Rewards** submodule is responsible for computing shaped reward signals for each agent at each tick, applying various incentive structures and safety guardrails. The main component is the `RewardEngine` class, which **computes per-agent rewards with clipping and guardrails** [24]. It aggregates multiple reward components – survival reward, need penalties, social interaction bonuses/penalties, work-related bonuses, and terminal penalties – then enforces limits like no positive reward after death and global clipping.

- **Base and Need-Based Rewards:** Each tick, the engine starts each agent's reward with a base survival reward (e.g. a small positive value each tick for being alive) as defined in config [25]. It then subtracts **need penalties** for hunger, hygiene, and energy deficits: for each need, if the need level is below 100%, a quadratic penalty (weighted by config coefficients) is applied to encourage agents to maintain their needs [26]. This results in a negative reward component growing with the square of the need deficit. These need-based adjustments ensure agents are internally penalized for neglecting basic needs.

- **Social Rewards:** If social features are enabled (controlled by a *social reward stage* flag in config), the engine adds rewards or penalties for social interactions [27]. Successful chats between agents yield a **bonus** for both speaker and listener based on a base value, plus modifiers proportional to the trust and familiarity between them [28]. Failed chat attempts can incur a small penalty for both parties [29]. These values come from config (e.g. `C1_chat_base`, `C1_coeff_trust`, etc.). Additionally, if the social stage allows it, the engine rewards agents for **conflict avoidance** events: when an agent successfully avoids a rivalry conflict (an event emitted by the world), a fixed bonus is granted [30]. The RewardEngine tracks recent social events by consuming them from the world (e.g. chat outcomes, avoidance triggers) and computes these bonuses via helper methods `_compute_chat_rewards` and `_compute_avoidance_rewards` [27] [31]. It also skips social rewards for agents in dire need states – if an agent's needs are extremely low (e.g. >85% deficit), `_needs_override` causes social chat rewards to be ignored for that agent [32], prioritizing survival behavior.

- **Work and Punctuality Rewards:** The engine incorporates **economic incentives**. Each tick, it adds a *wage reward* equal to the agent's wages earned minus wages withheld (unpaid) during that tick [33] [34], scaled by a wage rate config. This encourages agents to perform work to get paid. Similarly, it adds a **punctuality bonus** proportional to how timely the agent was for work (e.g. arriving on shift on time) [35]. This uses a `punctuality_bonus` factor from config and a punctuality metric tracked per agent (the world's context might increment this when the agent is on time). These components reward agents for good job performance and timeliness.

- **Terminal Penalties and Guardrails:** If an agent becomes **terminated** (e.g. "dies" or is removed from the sim) or is in the process of a forced exit, the engine applies a one-time **terminal penalty**. For example, if an agent "faints" (extreme hunger) or gets "evicted" (fired from job/

housing), a negative reward (penalty) is applied as specified in config (`faint_penalty`, `eviction_penalty`) [36]. The engine also maintains a *termination block window*: a short period after an agent's death during which the agent is disqualified from receiving any positive rewards. Concretely, if an agent died this tick or very recently, any positive reward that tick is zeroed out [37]. This prevents giving agents extra rewards in the moment of death (a guardrail against exploiting dying to collect rewards). After summing all components, the engine then **clips** the reward to a maximum magnitude per tick (and per episode) per config. It ensures the final reward is within ±`clip_per_tick` and accumulative rewards stay within an episode cap [38] [39]. If clipping occurs, it records the adjustment so that the net effect is traceable in logs.

- **Reward Breakdown and Logging:** The RewardEngine produces not just scalar rewards but also a **breakdown of components** for analysis. For each agent, it builds a dictionary of named reward components (survival, needs_penalty, social_bonus/penalty, wage, punctuality, terminal_penalty, etc.) and their values, as well as the total [40]. It stores this breakdown in `_latest_breakdown` each tick, and similarly keeps a log of recent social events in `_latest_social_events` [41]. These are used by the telemetry system to report reward composition and by the training loop for diagnostics. The interactions with configuration are significant: all weights, bonus values, and clip settings come from the `SimulationConfig` (`config.rewards` section), making the reward function highly tunable. **No major stubbed code is evident** – the RewardEngine appears fully implemented, though the exact values depend on config. One subtle point is the wage reward calculation uses `wages_paid - wages_withheld` without directly scaling by a rate inside `_compute_wage_bonus` [42] [34], implying the config's `wage_rate` might be factored into how wages are recorded rather than multiplied here. Overall, the Reward submodule provides a comprehensive shaped reward signal and ensures numeric stability and fairness through its guardrails.

# Scheduler Submodule (`townlet.scheduler`)

The **Scheduler** submodule manages timed events and perturbations in the simulation – essentially a system to schedule **random or scripted events** that affect the world over time. The primary class is `PerturbationScheduler`, which **injects bounded random events into the world** according to probability and cooldown rules [43]. These events, configured via `PerturbationSchedulerConfig`, include things like price spikes in the economy, utility outages (power/water blackouts), and arranged meetings between agents. The scheduler ensures such events occur in a controlled manner without overwhelming the simulation.

- **ScheduledPerturbation:** The submodule defines a dataclass `ScheduledPerturbation` that represents an event instance [44]. Each event has a unique `event_id`, a `spec_name` (referring to the type of event from the configuration), a `kind` (an enum `PerturbationKind` like PRICE_SPIKE, BLACKOUT, OUTAGE, ARRANGED_MEET), a start tick and end tick, plus any payload data and target agents involved. This data structure is used to keep track of events that are scheduled or currently active in the world.

- **Event Scheduling Logic:** The `PerturbationScheduler` maintains internal lists for events that are **pending** (scheduled to start in the future) and **active** (currently affecting the world). Each simulation tick, the scheduler's `tick()` method processes the timeline in several steps [45]:

- **Expire Active Events:** First it checks all active events and expires any whose `ends_at` tick has passed. Expiration triggers any cleanup needed (via `_on_event_concluded`) and removes the event from active list [46] [47].
- **Expire Cooldowns:** It then updates cooldown timers. There are cooldowns per event type and per agent to avoid repetitive triggers. Expired cooldown entries are dropped so those events/agents become eligible again [48].
- **Expire Window Counters:** If configured, it keeps a sliding window of recent events (to limit frequency of a given type). Events older than the window (e.g. last N ticks) are purged from the record [49].
- **Activate Pending Events:** Any pending events whose start time has arrived are moved into active state. The scheduler transfers them from the pending list to active list and calls `_activate()` for each [50] [51].

- **Maybe Schedule New Events:** Finally, it attempts to schedule new random events this tick. For each event spec in the config, it checks conditions via `_can_fire_spec` – e.g. not exceeding `max_concurrent_events`, respecting per-type cooldown, not exceeding event frequency window, and having a nonzero daily probability [52] [53]. If eligible, it draws a random chance; if it hits, a new event is generated via `_generate_event` and immediately activated [54] [55]. This may schedule at most one of each type per tick (loop breaks after scheduling if concurrency limit reached [56]).

- **Activating and Applying Events:** When an event is activated (either via reaching its start time or being scheduled immediately), `_activate()` moves it to the active dict and notifies the world. Activation involves:

- Adding a **cooldown** for that event's type (spec) so it won't trigger again for its cooldown duration plus a global cooldown buffer [57].
- Adding **per-agent cooldowns** for any agents targeted, preventing those agents from being targeted again for a configured number of ticks [58].
- Logging the event occurrence in the sliding window list (for frequency limiting) [59].
- Actually **applying the event's effects** to the world via `_apply_event` [60]. This method calls the appropriate world method depending on event kind:
  - For a **Price Spike**, it invokes `world.apply_price_spike(event_id, magnitude, targets)` to adjust resource prices, and emits a corresponding event message in the world (so telemetry knows prices spiked) [61].
  - For **Blackout/Outage**, it calls `world.apply_utility_outage(event_id, utility)` (utility could be "power" or "water") to cut service, and emits an event describing which utility went down [62] [63].
  - For **Arranged Meet**, it calls `world.apply_arranged_meet(location, targets)` which might force specific agents to meet at a location, then emits an event indicating the meeting was scheduled [64].
- Emitting a **"perturbation_started"** event to the world's event log with details (ID, type, targets, end time) [65]. The world (or telemetry) can use this for logging or UI notifications.
- When events end naturally, `_on_event_concluded` is called to reverse their effects if needed (e.g. clear a price spike or restore utilities) and emit a "perturbation_ended" event [66] [67].

- The scheduler also provides methods for external control: `schedule_manual(...)` to inject an event on demand (used by console commands or tests to force an event) [68] [69], and `cancel_event(...)` to abort a pending or active event by ID [70] [71], both of which also notify the world of cancellation [72] [73].

- **State Persistence:** The PerturbationScheduler tracks state to allow saving and restoring. It can export its internal state (active events, pending events, cooldown timers, RNG state, etc.) to a dictionary [74] [75]. Conversely, `import_state` can restore those, making the system robust to simulation pause/resume or snapshotting. It also supports seeding its random generator for deterministic sequences if needed [76].

**Gaps or Stubs:** The scheduler implementation is quite complete and by design largely driven by configuration. One potential ambiguity is that it relies on the `WorldState` to provide certain methods (like `apply_price_spike`, `find_nearest_object_of_type`, etc.). These hooks must be implemented in the world; if not, the events might do nothing. In general, however, the code for common perturbations is present. Another point is the system's flexibility: it currently supports a fixed set of event types. If new event kinds are needed, the `_apply_event` and `_generate_event` methods would need extension. The existing code already covers random duration selection within a range, random target agent selection for meets (ensuring at least two eligible agents or else the event is not scheduled) [77], and enforces a max events per window. Overall, the **Scheduler** coordinates time-based world changes, ensuring the simulation experiences dynamic events like crises and meetings in a controlled fashion.

# Lifecycle Submodule (`townlet.lifecycle`)

The **Lifecycle** submodule handles agent life-cycle management – spawning new agents, removing (terminating) agents, and enforcing rules like mortality and employment turnover. The key class is `LifecycleManager`, which centralizes **lifecycle checks for agent exits and respawns** [78]. This subsystem ensures that agents who “die” or otherwise exit are removed properly and replaced if appropriate, and that certain simulation phases (like daily job evaluations) are processed.

- **Mortality and Hunger:** The lifecycle manager monitors agents’ vital conditions each tick via `evaluate()`. For each agent, it checks if the agent’s hunger need has dropped too low. By default, if hunger $\leq$ 0.03 (3% of full) and mortality is enabled, the agent is considered to have **fainted/died** [79]. The manager then marks that agent as terminated with reason `"faint"` [79]. (The threshold and toggle come from config: `mortality_enabled` can be turned off to disable death from hunger [80] [81].) All other agents not meeting a termination condition are marked as continuing (False for terminated) by default [82]. This simple mortality rule is an implementation of a health-based end-of-life: currently hunger is the only need that triggers death, which may be a designed simplification or an area to extend (e.g. hygiene or energy could potentially be considered in future).

- **Employment and Eviction:** The `LifecycleManager` also enforces an **employment life-cycle** if configured. If `config.employment.enforce_job_loop` is True, `evaluate()` will call an internal `_evaluate_employment()` each day to handle job exits [83]. In these checks, the manager looks for agents who have been absent from work excessively or who are queued for firing:

- It resets a daily counter at the start of a new day (based on tick and a configured day length) [84].
- Any agent that has accumulated too many absent shifts (e.g. >= `max_absent_shifts` in 7 days) is added to an exit queue via `world._employment_enqueue_exit` [85].
- Agents can also be manually flagged for removal (e.g. an external approval) – those appear in `world._employment_manual_exits` and are immediately processed as leaving with reason "manual_approve" [86].

- Each tick, for agents in the exit queue, it checks how long they've been waiting. If an agent has been in the queue longer than the review window (e.g. a week of simulation ticks), the manager executes their exit automatically with reason "auto_review" [87].
- It also enforces a **daily cap** on how many agents can be removed per day. After handling mandatory exits, it will remove up to `daily_exit_cap` agents from the queue (oldest first) each day with reason "daily_cap", to simulate limited firings per day [88].

- When an agent is processed for exit, `_employment_execute_exit` is called: it clears that agent from the queue, resets their absence counter, marks that agent's `exit_pending` flag off, and emits an event `"employment_exit_processed"` with details (agent, job, reason, tick) [89] [90]. It also logs the termination reason as "eviction" internally [91]. In effect, "eviction" in this context means the agent is removed from the simulation for job-related reasons (fired or left town). This dual mechanism of fainting (hunger death) vs. eviction (job removal) constitutes the two main termination pathways.

- **Finalizing Exits and Respawns:** Once `evaluate()` identifies which agents should terminate (with a boolean map and reasons), the `LifecycleManager.finalize(world, tick, terminated)` is called at the end of the tick to carry out the removals [92]. For each agent marked terminated:

- It calls `world.remove_agent(agent_id, tick)` to remove the agent from the simulation and retrieve that agent's **blueprint** (a record of the agent's attributes/state) for potential respawn [93]. The blueprint typically contains initial data needed to recreate the agent.
- If a blueprint is returned (meaning the agent can be respawned), the manager prepares to respawn a new agent in place of the old one. It determines an `origin` (the original agent's ID or origin ID if already respawned before) to carry continuity, then requests the world to generate a new unique agent ID for the offspring/replacement [94]. It updates the blueprint with this new agent ID and records the original ID for reference.
- It schedules a respawn by creating a **_RespawnTicket** (an internal dataclass with the new agent's blueprint and the tick at which to spawn) after a delay configured by `respawn_delay_ticks` [95] [96]. For example, if `respawn_delay_ticks` is 0, the agent might respawn next tick; if it's >0, the agent will be absent for that many ticks.
- The ticket is added to a `_pending_respawns` list. Then the old agent is fully removed from the world (so agents list shrinks immediately).

On subsequent ticks, the manager runs `process_respawns(world, tick)` at the start of the tick to check if any pending respawn's scheduled time has arrived [97]. For each due ticket, it calls `world.respawn_agent(blueprint)` to create a new agent in the world using the saved blueprint [98]. This mechanism ensures population turnover: whenever an agent exits (by death or eviction), a new one comes in after a short delay, keeping the number of agents stable if the scenario expects it. If no respawns are pending or not yet due, the tickets remain in the list for future ticks.

- **Other Controls and State:** The manager tracks some counters like `exits_today` (how many agents were removed in the current day, for the daily cap) and uses `_employment_day` to know when a new day starts for resetting counts [99] [84]. It provides methods to change parameters at runtime: `set_respawn_delay(ticks)` to adjust respawn delay dynamically [100], and `set_mortality_enabled(flag)` to toggle hunger-death on or off [101] (the console can invoke these, e.g. via a `toggle_mortality` command [81] [102]). The manager also can export minimal state (like exits_today count) for snapshotting and restore them to continue simulation smoothly [103].

**Gaps or Future Work:** The lifecycle rules are primarily about hunger and job exit; other aspects (illness, old age, etc.) are not modeled, which could be an area for extension. The code references a conceptual design snapshot, suggesting the current implementation covers the core points, but further "stages" might be planned. For example, the `mortality_enabled` flag hints at possibly turning off death for certain experiments. The employment exit process is relatively complex and tightly integrated with the world's internal queues for exits; it assumes the world properly tracks absences and queueing. No obvious stubbed code is present; all major functions have implementations. One should note that `world.remove_agent` must provide a blueprint for respawn – if it returns None, the manager will skip respawn for that agent [104]. In summary, the Lifecycle submodule robustly manages agent turnover, ensuring the simulation can run indefinitely with agents cycling in and out according to the rules.

# Console Interface (`townlet.console`)

The **Console** submodule provides a developer/admin **command-line interface** to the simulation, enabling live debugging, state inspection, and manipulation of the running environment. It essentially defines a set of textual commands and their handlers which can be invoked (for example, via a UI or API) to query the simulation or perform privileged actions. The design is modular: commands are parsed into a uniform format and then dispatched to appropriate handler functions.

- **Command Parsing and Routing:** The console defines a data class `ConsoleCommand` that represents a parsed command with a name, arguments, and keyword args [105]. When a command comes in (as raw text or JSON), it is first authorized (the console supports an auth mechanism for security, not detailed here) and then packaged into a `ConsoleCommand`. The `ConsoleRouter` holds a registry of command names to handler callables [106]. Calling `dispatch(command)` will look up the command's name and call the corresponding handler, raising an error if the name is unknown [107]. This design makes it easy to add new console commands by writing a handler function and registering it.

- **Telemetry Bridge:** Many console commands are read-only queries that retrieve simulation metrics. For convenience, the console uses a `TelemetryBridge` class that interfaces with the `TelemetryPublisher` to fetch the latest snapshots of various data [108] [109]. For example, the bridge's `snapshot()` returns a comprehensive dictionary of all current telemetry (jobs, economy, relationships, events, etc.) in a single call [110] [111]. There are also helper methods like `relationship_summary_payload()` to format friends/rivals lists per agent [112] [113] and `relationship_detail_payload(agent_id)` to get detailed relationship stats for one agent [114] [115]. This bridge ensures console commands can pull the most up-to-date info without duplicating telemetry logic.

- **Supported Commands:** The console defines a wide array of **handler functions** for different commands, covering both state inspection and modifying actions:

- *Simulation/Telemetry Status:* Commands like `"telemetry"` return a full telemetry snapshot (via the bridge) [116]. `"health_status"` returns a summary of internal health metrics (tick performance and telemetry queue stats) [117]. `"employment_status"` gives employment metrics and pending exits [118].
- *Social Relations:* `"relationship_summary"` provides each agent's top friends and rivals plus churn stats [119]. `"relationship_detail <agent>"` gives a deep dive into one agent's

relationship ledger and recent changes [120] [121]. `"social_events"` lists recent chat and avoidance events, with an optional limit parameter [122] [123].
- *World State:* `"affordance_status"` inspects the status of affordance execution (running interactions) and reservations in the world [124] [125]. `"queue_inspect <object_id>"` returns the state of a specific object's queue: which agents are waiting, their join ticks, any cooldowns, and how many times the queue has stalled [126] [127]. `"conflict_status"` compiles info on conflict metrics – queue events, rivalry snapshots, recent rivalry events, stability alerts – to debug the social conflict system [128] [129].
- *Lifecycle Controls:* `"employment_exit review"` shows the current exit queue (who is pending firing) [130] [131]; `"employment_exit approve <agent>"` or `"defer <agent>"` let the user manually approve or postpone a particular agent's firing [132]. `"possess <agent_id>"` allows manual control of an agent: "acquire" possession to remove it from AI control, or "release" to hand it back [133] [134]. This sets a flag so the policy will no longer decide actions for that agent, effectively allowing a human or external policy to step in [135] [136]. `"kill <agent_id>"` immediately removes an agent from the world [137] (calling `world.kill_agent()` internally). There's also `"toggle_mortality true|false"` to enable/disable agent death by hunger on the fly (toggling the lifecycle manager's flag) [81] [102]. `"set_exit_cap N"` adjusts the daily job exit cap in the config at runtime [138], and `"set_spawn_delay T"` changes the respawn delay ticks for new agents [139] [140]. These commands provide live control over lifecycle parameters.
- *Perturbation Controls:* The console can trigger or inspect scheduled events. `"perturbation_queue"` returns the scheduler's current pending/active events and cooldowns [141] [142]. `"perturbation_trigger <spec> ..."` allows injecting an event of a given type with optional parameters like start delay, duration, targets, magnitude, etc. [143] [144]. This uses the scheduler's `schedule_manual` under the hood and returns an acknowledgment with the event details [145] [146]. `"perturbation_cancel <event_id>"` can cancel a scheduled or ongoing event by ID [147] [148].
- *Snapshots and Debug:* `"snapshot_inspect <path>"` reads a saved simulation snapshot file and returns high-level info (schema version, tick, config ID) without loading it fully [149] [150]. `"snapshot_validate <path>"` attempts to load a snapshot with the current config (optionally strictly) to check if it's compatible or needs migration [151] [152]. `"snapshot_migrate <path>` will load and re-save a snapshot to the latest version, optionally to a different directory [153]. These help in managing simulation state across versions.
- *Policy and Training:* There are commands to inspect and control the RL policy aspect. `"promotion_status"` returns the current status of the policy promotion system (which tracks if a new policy is ready to be promoted) [154]. `"promote_policy <checkpoint> [--policy-hash H]"` marks a new trained policy (given by checkpoint path and optional hash) as the active one (this ties into the **PromotionManager** which handles rolling out new policies) [155] [156]. Similarly, `"rollback_policy [--checkpoint X] [--reason R]"` will revert to the previous policy version, optionally specifying a checkpoint to roll back to and logging a reason [157] [158]. `"policy_swap <checkpoint>"` is another developer command to manually swap the current policy network with a given checkpoint (used for testing policy hot-reloading) [159] [160]. These commands are more administrative and reflect the training integration in the system.

This extensive list (not exhaustive here) illustrates that the console is essentially a **debugging and control dashboard**, with commands acting like an admin API. Each handler function validates inputs and interacts with the corresponding subsystem (world, lifecycle, scheduler, telemetry, etc.), then returns a JSON-serializable result (often including an `"error"` field if something goes wrong or is misused) so that the caller can display or log the outcome.

- **Integration with Simulation Loop:** The console works in tandem with the telemetry system to execute commands. The `TelemetryPublisher` holds a queue (`_console_buffer`) of incoming command envelopes. External tools (like a web UI or script) can inject console commands via `TelemetryPublisher.queue_console_command(...)`, which will authenticate and queue the command [161] [162]. On each tick, the simulation loop drains this buffer at the beginning of the tick (SimulationLoop.step calls `console_ops = telemetry.drain_console_buffer()` to grab any pending commands) [163]. The world then processes these console operations via `world.apply_console(console_ops)` and returns results which telemetry records [164] [165]. In practice, `world.apply_console` likely iterates through the commands, uses the ConsoleRouter to dispatch them, and collects their results in a list. The telemetry publisher then stores the results (in `_console_results_history`) and makes them available via `latest_console_results` for any external observer [166] [167]. This design ensures console commands are applied **synchronously within the simulation tick**, so their effects (if any) are reflected immediately in that tick's world state and telemetry. Console actions like killing an agent or toggling a flag happen in a controlled point in the loop (right after respawns and before the main agent decisions).

**Status and Gaps:** The console submodule appears fully implemented with a wide range of commands. Its docstring calls it a "validation scaffolding" [168], which suggests it was built to facilitate testing and debugging during development. There are no obvious stub functions – each declared command has a concrete handler. One minor aspect is the security model: it references `ConsoleAuthenticator` for authorisation [169] [162], meaning in practice an auth token might be required (the specifics of `console_auth` config are not detailed in the snippet). From a design perspective, any new simulation feature likely needs a corresponding console command for inspection or toggling – the current set covers relationships, employment, conflicts, etc., indicating completeness for those domains. The console greatly aids in debugging and controlling the simulation in real-time, working hand-in-hand with telemetry to retrieve the latest data.

# Telemetry and Metrics Submodule (`townlet.telemetry`)

The **Telemetry** submodule is responsible for **collecting, aggregating, and publishing simulation metrics and events**, as well as relaying console commands/results. It acts as the central data pipeline that observes the simulation state each tick and outputs useful information for monitoring or training. The main class, `TelemetryPublisher`, **publishes observer snapshots and consumes console commands** [170], effectively bridging the simulation loop with external systems or logs.

- **Metric Collection Per Tick:** Every simulation tick, the SimulationLoop calls `telemetry.publish_tick(...)` with the current tick number and a host of data: the world state, the latest observations and rewards, any events emitted by the world, policy snapshots, reward breakdowns, stability metrics, perturbation state, etc. [171] [172]. The `TelemetryPublisher.publish_tick` method then **updates its internal records** for all these categories:
- It queries the world for **queue metrics** (like queue lengths, and internal counters such as cooldown events, ghost steps, rotation events) and stores them, also computing a delta since

last tick to measure new queue fairness events [173] [174]. It appends this data to a history list (capped length) to track recent queue metrics over time.

- It gets a **rivalry snapshot** from the world (if available) – essentially the matrix of rivalry intensities between agents – and stores it [175] [176]. It also consumes any **new rivalry events** (spikes in rivalry or conflicts resolved) from the world and appends them to a history, similar to queue events [177] [178].
- It calls the world for a **relationship metrics snapshot** (aggregate stats about relationships) and a full **relationship tie snapshot** (pairwise trust/familiarity/rivalry values). With these, it computes updates since the previous tick to identify which relationships changed (for logging) [179] [180]. It then produces a summarized view (e.g. top friends/rivals per agent, and churn counts) and stores that as `_latest_relationship_summary` [181] [182].
- It collects **embedding allocator metrics** from the world (how many embedding slots used, etc.) and stores them for analysis [183].
- Any raw **events** passed in (those emitted by the world this tick) are recorded. The TelemetryPublisher maintains `_latest_events` which include things like affordance events (start/finish/fail of actions), console events, perturbation events, etc. [184]. It also filters those to capture specific categories, for example:
    - It extracts any **affordance precondition failures** (cases where an agent tried an action but a precondition failed) to a separate list, for debugging why agents couldn't act [185].
    - It passes events along with social events to a narration processor to possibly generate or throttle narrative text (not detailed here) [186].
    - It notifies any **event subscribers** (like the console's `EventStream` listener) by calling their callback with the latest events [187].
- If provided, it updates the latest **policy snapshot** (statistics from the policy like loss, entropy, etc. per agent) and **possessed agents** list (which agents are under manual control) [188] [189]. It also notes the current policy identity (a hash or ID of the policy in use).
- It builds a **"job snapshot"** for all agents, collating each agent's employment-related data: job ID, on/off shift status, wallet, lateness counter, performance stats (meals cooked/consumed, etc.), shift state, attendance ratio, current consecutive late ticks, absences in 7 days, wages withheld, and whether they are flagged for exit [190] [191]. Essentially, this is a per-agent dictionary summarizing their current work and need status, used for monitoring training or experiment KPIs.
- It similarly builds an **economy snapshot** of all objects in the world (each object's type and current stock of items) [192], and captures **economy settings** from config (like global prices of goods or tax rates, if any) [193].
- It keeps track of **perturbation state** by querying the scheduler's latest state (active events, etc.) and storing it [194].
- It logs **stability metrics** from the StabilityMonitor (like any alert flags, aggregate metrics computed for system stability) and the current **promotion state** from PromotionManager if present (this might include info on whether a new policy has "graduated" in training) [195].
- **Health metrics** (tick duration, telemetry queue length, etc.) are recorded each tick as well [196].

All these updated metrics are stored in the TelemetryPublisher's attributes prefixed with `_latest_...` (e.g. `_latest_queue_metrics`, `_latest_conflict_snapshot`, `_latest_relationship_updates`, `_latest_reward_breakdown`, `_latest_stability_metrics`, etc.). They represent the authoritative source of truth for the simulation's state from a monitoring perspective.

- **Publishing and Transport:** The TelemetryPublisher can be configured to output this data to some external sink. It supports a **transport mechanism** (e.g. HTTP, file, or Kafka) defined by `config.telemetry.transport`. In the code, it creates a `TransportBuffer` and a background thread that periodically flushes telemetry data out [197] [198]. Each tick's data can be formatted (likely as JSON) and sent. The specifics aren't fully shown, but we see it maintains `_transport_status` (connected, errors, queue length, etc.) [199] and starts a thread named `"telemetry-flush"` on init [200]. This suggests the telemetry system continuously streams data off-simulation so that training processes or dashboards can consume it in near real-time. If transport fails or is not configured, data can still be accessed via the console or logs.

- **Console Command Buffer:** As noted, TelemetryPublisher also acts as the bridge for console commands. It has a `_console_buffer` list that collects incoming `ConsoleCommandEnvelope` objects (each containing a command and metadata like issuer) [161]. When an external interface calls `queue_console_command(cmd)`, the TelemetryPublisher first authenticates it (using the `ConsoleAuthenticator` and config-provided token/credentials) [162]. If authorized, it sanitizes the command and appends it to `_console_buffer`. At the start of each tick, `drain_console_buffer()` is called to retrieve and clear all pending commands [201], which are then given to the world to execute. After the world runs them via the ConsoleRouter, the results (success or error responses) are passed back to TelemetryPublisher using `record_console_results(results)` [202]. The telemetry keeps a history of console results (up to e.g. 200 last commands) and makes the latest available to the TelemetryBridge (so that a UI can query what happened when a command was run) [203] [204]. This integration ensures console operations are part of the telemetry stream – for instance, one can see in telemetry output what console commands were executed and their outcomes.

- **Alerts and Narratives:** TelemetryPublisher also includes some higher-level monitoring:

- It uses a `NarrationRateLimiter` to throttle how often narrative events (like relationship change narrations) are emitted [205]. It collects `_latest_narrations` and a `_latest_narration_state` if any narrative system is active.
- It captures stability alerts (from StabilityMonitor) and queue fairness history to potentially raise warnings if the simulation is unstable or agents are stuck, etc. [206] [207].
- The PromotionManager integration (accessible via telemetry) allows telemetry to note if a training policy promotion happened and what the current release is – these appear in telemetry outputs as well [208].

**Use in Training:** The telemetry outputs are crucial for training analysis. For example, the `reward_breakdown` and `social_events` logged each tick are used to calculate training rewards and to debug agent behavior. The training harness can also retrieve cumulative metrics from telemetry if needed (though primarily it uses its own tracking for loss, etc.). The telemetry data (like the conflict snapshot and relationship summary) is also used in *curriculum learning* or adaptive systems: the console and training code reference these to adjust difficulty or to determine when to promote a policy version.

**Gaps and Ambiguities:** The Telemetry system is quite detailed. One area to note is performance – it collects a lot of data each tick, so in a very large simulation this could become a bottleneck, but the design allows toggling certain features off via config (e.g. one could set social reward stage to OFF to skip some metrics, or tune what events world emits). The schema version (`schema_version = "0.9.7"`) [209] suggests the telemetry format is in a 0.9.x development stage, and indeed many fields are geared toward evolving requirements. Some fields like `latest_job_snapshot` or `latest_conflict_snapshot` are composite and could be subject to change as the design iterates. There is no obvious stub code; all metrics listed in config (queues, relationships, etc.) have corresponding capture code. The TelemetryTransport might not be fully implemented or may rely on external configuration – if misconfigured, telemetry would still function locally but not send out data (the code logs transport errors but continues). In summary, the Telemetry submodule provides a comprehensive mirror of the simulation's state and health, and it underpins both the **monitoring** (for developers/operators) and the **training feedback loop** (for reinforcement learning).

# Training & Replay Code

In addition to the core simulation modules, Townlet includes top-level code for training machine learning policies and for replaying recorded simulation data. This encompasses command-line scripts (in `scripts/`) and the `townlet.policy` package, which together form the **Authoritative Design Reference for training workflows**. The training system is designed to accommodate multiple modes (reinforcement learning from live rollouts, offline learning from replays, behavior cloning, etc.) and to integrate with the simulation loop.

- **Training Entry Point:** The primary entry-point is the script `run_training.py`, which provides a CLI to configure and launch training runs [210]. Users specify a YAML config file for the simulation and optional overrides. The `--mode` flag selects the training mode: `"replay"` (learn purely from a fixed dataset of observations), `"rollout"` (learn by running the sim and collecting new experiences), `"mixed"` (a combination), `"bc"` (behavior cloning from trajectories), or `"anneal"` (a staged training starting with BC then annealing into RL) [211]. There are further options to supply replay samples or manifests, to capture rollout data, and to tweak PPO hyperparameters on the fly [212] [213]. This flexible CLI indicates that Townlet's training can be conducted in various ways depending on experiment needs.

- **TrainingHarness and Coordination:** Internally, `run_training.py` loads the simulation config (which includes training settings) and instantiates a `TrainingHarness` from `townlet.policy.runner` [214] [215]. The **TrainingHarness** is responsible for **coordinating RL training sessions** [216]. It interprets the `config.training.source` to decide what procedure to run. For example:

- If the mode is **behavior cloning (bc)**, it invokes `run_bc_training()`, which will load a dataset of demonstration trajectories and train a supervised policy network [217] [218].
- If the mode is **anneal**, it calls `run_anneal()`, which presumably first does BC then gradually mixes in RL (the code sets up an anneal context and baselines for mixing) [219] [217].
- For pure RL modes, the harness's `run()` currently defers to the CLI to handle `"rollout"` or `"replay"` via specialized calls (the harness raises NotImplemented for those in `run()` to force using the explicit methods, as a design choice) [220]. Instead, the CLI script directly calls methods like `harness.run_replay()`, `run_replay_dataset()`, or `run_rollout_ppo()` based on flags.

The TrainingHarness keeps track of training state (like current PPO training step, learning rate schedule in `_ppo_state`) and handles things like applying different social reward stages or anneal ratios at certain cycles [221] [222]. It also contains a **PromotionManager** instance [223], which is used to decide when a trained policy is "good enough" to be promoted (this ties into the console commands above and stability monitoring).

- **Replay Dataset and Sample Handling:** A notable part of the training pipeline is the **replay data handling**. The `townlet.policy.replay` module provides utilities to load and manipulate recorded observation-action data:
- **ReplaySample**: a dataclass representing a sequence of timesteps from one agent's perspective, including the observation `map` **tensor,** `features` **vector**, and the corresponding actions, log probabilities, value predictions, rewards, and done flags for each timestep [224] [225]. It also carries

metadata (feature names, etc.). The `ReplaySample` class validates that all arrays have consistent lengths and shapes on initialization [226] [227] and automatically adjusts dimensions (e.g. adds time dimension if a single timestep) [228] [229]. It ensures crucial features for conflict metrics (like rivalry stats) are present in the metadata, throwing errors if not [230] [231] – this is because some training analyses (like curriculum decisions) rely on those stats [232].

- **ReplayBatch**: a dataclass for batching multiple samples (for training mini-batches). It stacks arrays from multiple ReplaySamples and again checks dimension consistency (ensuring each sample had the same timestep count if drop_last is false, etc.) [233] [234]. It also provides a `conflict_stats()` method that computes mean and max of rivalry features across the batch [232]. The purpose is to feed this batch data into the policy network for training (e.g. computing loss on a batch of trajectories).
- **Loading Replays**: The function `load_replay_sample(path, meta_path)` loads a `.npz` file containing saved arrays (`map`, `features`, `actions`, etc.) and an optional JSON metadata file [235] [236]. It verifies all required fields are present and returns a ReplaySample [237] [238]. Additionally, the code supports **manifests**: a manifest file (JSON or YAML) can list multiple samples with their file paths, which `_load_manifest` will parse into a list of (sample_path, meta_path) pairs [239] [240]. This allows easy bundling of a dataset.
- **ReplayDataset**: an iterable that yields ReplayBatch objects. It takes a `ReplayDatasetConfig` which contains either a list of entries or pointers to a manifest or capture directory [241] [242]. The config can specify batch size, whether to shuffle each epoch, and whether to stream from disk or preload into memory. The ReplayDataset, upon initialization, will either load all samples into memory or prepare to load on-the-fly if streaming [243] [244]. It checks that all samples are homogeneous in shape (so they can be batched) [245] [246] and calculates baseline metrics across the dataset (like average values of certain metrics) [247] [248]. Iterating over ReplayDataset yields batches of the configured size, shuffling if required [249] [250]. This is used for training loops – e.g., feeding each batch into PPO update or evaluation of performance.

The TrainingHarness provides convenience methods like `run_replay(sample_path)` which simply loads one sample and prints its conflict stats [251], or `run_replay_batch(list_of_pairs)` to load multiple samples into one batch and summarize [252]. More importantly, `run_replay_dataset(config)` will iterate through an entire dataset (possibly multiple batches) and print stats for each, returning an aggregate summary [253]. These are useful for diagnostics or for computing curriculum signals (e.g., conflict intensity in samples).

- **Rollout (On-Policy) Training:** The harness also supports collecting fresh data via simulation **rollouts**:
- `capture_rollout(ticks, ...)` will initialize a SimulationLoop, optionally auto-inject some default agents if none (to ensure something to simulate) [254], then step the simulation for the given number of ticks [255]. On each tick, it calls `loop.policy.collect_trajectory(clear=True)` to grab the trajectory frames (observations, actions, etc.) since last collection, and appends them to a `RolloutBuffer` [255]. It also records any events that happened that tick into the buffer (these events could include outcomes needed for reward shaping or analysis) [256]. After running, the RolloutBuffer contains a list of frames and can be saved to disk (the code provides `buffer.save(output_dir)` functionality) [257]. This essentially automates playing out the environment to generate training data.
- `run_rollout_ppo(ticks, batch_size, epochs, ...)` combines rollout capture and PPO training in one go. It first triggers a rollout of `ticks` length to produce a RolloutBuffer [222]. Then it converts that buffer into a ReplayDataset (using `buffer.build_dataset`) and calls `run_ppo` on it [258]. The `run_ppo` method (not fully shown) would iterate for the specified

number of epochs, performing the PPO optimization on the collected data. The CLI flags like `--epochs`, `--ppo-learning-rate`, etc., are passed into this call via the harness.

- We see in the code that before each rollout PPO cycle, the harness might adjust the social reward stage: `self._apply_social_reward_stage(next_cycle)` is called to potentially make the environment harder or more realistic as training progresses [259] [222]. This indicates a curriculum where early training might turn off some complex social rewards and later enable them (for instance, stage C1, C2 as referred in RewardEngine) as cycles increase.

- **Behavior Cloning and Annealing:** The harness's `run_bc_training` method handles supervised learning. It loads a dataset of trajectories (from a manifest) and uses a **BCTrainer** (defined in `townlet.policy.bc`) to train a network on that dataset [260] [261]. It requires PyTorch – if `torch_available()` returns False, it will raise an error that PyTorch is needed for BC [262]. This check means on systems without the ML framework, BC training can't run (a designed limitation or a way to fail early). The BCTrainer likely trains a `ConflictAwarePolicyNetwork` (which is the model architecture that takes features & map and outputs actions) using the dataset. After training, metrics (accuracy, loss, etc.) are returned [261]. In **anneal mode**, BC training would be followed by an RL phase: the harness's `run_anneal()` (not fully shown) would presumably call run_bc_training then use the resulting model as initialization for PPO, gradually mixing in the on-policy loss. The harness tracks an `_anneal_ratio` to blend BC vs RL objectives and monitors performance to decide when to end annealing [263] [217].

- **Policy and Promotion:** The training code uses a **ConflictAwarePolicyNetwork** as the underlying model (imported from `policy.models`). This suggests the neural network is specially designed to handle the conflict features and possibly multi-modal inputs (grid + feature vector). The training uses PPO algorithms (the `policy.ppo.utils` module is imported for GAE, loss functions, etc. [264]). After some training cycles, the PromotionManager can be used to test if the new policy performs better (perhaps by running evaluation rollouts or checking stability metrics). If conditions meet, the harness (or an external process) can mark the new policy as promoted, which via the console command can swap the active policy in the running sim. This is a sophisticated setup aimed at continual learning or curriculum learning in a live environment.

**Overall,** the Training and Replay components of Townlet provide a powerful framework to train AI agents in the simulation. They tightly integrate with the simulation loop: replays use recorded data consistent with the ObservationBuilder's output, and live rollouts use the same SimulationLoop but with a **PolicyRuntime** that interacts with the learning algorithm. Notably, the **PolicyRuntime** (in `policy.runner.PolicyRuntime`) acts as the glue during live simulation, deciding actions via either scripted behavior or a neural policy, and collecting trajectories for learning [265] [266]. It also handles things like option commit (ensuring an agent sticks to a chosen high-level action for a few ticks) and blending between scripted and learned behavior (anneal blending) [267] [268]. These aspects are configured via the training config (e.g., `anneal_enable_policy_blend`).

**Gaps or Ambiguities in Training:** The training code is quite extensive, but some parts are marked as *scaffolding*. For example, `PolicyRuntime` and `TrainingHarness` have elements to accommodate complex training schemes, and some methods in `TrainingHarness` (like run for certain modes) delegate to scripts rather than doing everything internally [220]. This suggests some features might still be in development or expected to be orchestrated externally rather than fully automated. The integration with actual ML frameworks (Torch) is present (model definitions, training loops), and errors are raised if unavailable, indicating the intention to use these models. One area of ambiguity is how

**promotion** is decided – it likely relies on stability metrics or manual triggers to replace the running policy. Another is the `"mixed"` mode which is mentioned in CLI but not explicitly handled in harness code we saw (possibly treated similarly to anneal or rollout modes). Despite these, the training system appears functional, enabling a lifecycle from **data collection** (rollouts or replays) -> **model training (PPO/BC)** -> **policy evaluation/promotion**, all while interfacing with the simulation's config and telemetry.

---


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 builder.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/observations/builder.py

24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 engine.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/rewards/engine.py

43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 perturbations.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/scheduler/perturbations.py

78 79 80 82 83 84 85 86 87 88 89 90 91 93 94 95 96 97 98 99 100 101 103 104 manager.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/lifecycle/manager.py

81 102 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 168 203 204 handlers.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/console/handlers.py

92 163 164 165 171 172 194 195 196 206 207 208 sim_loop.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/core/sim_loop.py

161 162 166 167 169 170 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 197 198 199 200 201 202 205 209 publisher.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/telemetry/publisher.py

210 211 212 213 214 215 run_training.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/scripts/run_training.py

216 217 218 219 220 221 222 223 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 runner.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/policy/runner.py

224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 replay.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/policy/replay.py