ChatGPT

Townlet Design Audit Reference

Design Intent & Documentation Overview

Project Vision & Goals

Townlet's vision is to deliver an **"always-on small-town simulation"** with a handful of DRL (deep reinforcement learning) agents living out daily routines. The aim is both entertainment and a proof-of-concept for configurable, observable RL sandboxes in a social environment [1]. In concrete terms, the project strives to sustain agents' needs, employment, and simple social dynamics without collapse through early milestones (self-sufficiency >70% agents, job tenure ≥3 days, etc.), then layer on more complex social features in later phases [2]. Success is measured by key metrics (KPIs) such as maintaining agents' basic needs above thresholds over 24h periods, achieving punctual attendance rates, forming social ties (trust/rivalry), and meeting performance targets like **≥4,000 env-steps/sec** in training [2] [3]. The scope is phased (A, B, C): Phase A covers core needs, jobs, and basic console/UI; Phase B introduces relationships in observations; Phase C adds social rewards, conflict mechanics, and agent behaviour cloning/annealing [4]. Non-goals include heavy production features like 3D rendering or monetization loops – the focus remains on the simulation and AI behavior itself [5]. Overall, the goal is a stable, extensible town life sim that can be *observed and nudged by humans* (via a console and dashboards) while the agents learn to behave plausibly in an emergent, story-worthy way [6] [7].

Functional Requirements

Townlet's functional requirements capture the capabilities the system **must** deliver. At a high level, the simulation must run on a fixed tick cadence (250 ms per tick, i.e. 4 ticks/sec real-time) and be *deterministic* given a seed [8]. Time advances at 1,000 ticks per in-sim day by default [8], with support for time dilation speeds and pauses. The world should support **hot-reloadable configurations** and feature flags that enable or disable systems in stages [9] – for example, toggling the relationship system on only when reaching Phase B, or social rewards only in Phase C. Agents perceive the world through one of multiple observation modes ("full", "hybrid", "compact"), each providing a different level of spatial detail, along with a shared bundle of scalar features like needs, wallet, job status, and recent actions [10] [11]. The action space is hierarchical: agents have low-level primitives (move, interact, wait, talk, buy, etc.) and higher-level options (e.g. GetFed, GoToWork, Socialise) that commit them to multi-tick behaviours [12] [13]. An **action masking** mechanism must prevent impossible actions (e.g. "Talk" when no target nearby) by providing a mask to the RL policy [14] [15].

The simulation enforces realistic **affordances** (interactions) and physical constraints: interactive objects (beds, fridges, shops, etc.) define what actions they afford via YAML specs, including preconditions, durations, and effects on agent state [16] [17]. Only one agent can use a given object or tile at a time – thus a reservation and queuing system is required. The **queue/reservation manager** must handle agents lining up for objects with defined queue nodes, apply tie-break rules when agents arrive simultaneously (by a "patience" trait or random jitter), and implement fairness policies (cooldowns for cutting in and a slight priority boost for longer-waiting agents) [18]. It also needs a deadlock resolution: after repeated stalemates, allow a one-tick ghost movement to break the impasse and log the incident [19]. The simulation must manage **agent needs** (hunger, hygiene, energy) that decay over time [20], **jobs** with scheduled shifts that pay wages, and simple economics (prices of goods, daily rent, etc.) to create pressure on agents to work [16] [21]. Fail-safes are required: for instance, if an agent's needs drop too low or they miss too many shifts, the **Lifecycle Manager** should remove (terminate) that agent and eventually spawn a replacement, maintaining population continuity [22] [23]. All removals are governed by exit criteria (e.g. max two exits per day, cooldown between spawns) to avoid rapid cascade failures.

Reward shaping is a first-class concern: each tick, agents receive a shaped reward signal combining *homeostatic penalties* for needs (e.g. quadratic cost for hunger/hygiene deficits), positive rewards for working (wage accrual per tick and punctuality bonuses for arriving on time), a small survival bonus each tick, and (in later stages) social interaction bonuses [24] [25]. The design demands *strict reward guardrails*: clip the per-tick reward magnitude and per-episode cumulative reward to configured bounds, and **suppress any positive reward around death/exit events** so agents cannot exploit dying to "escape" negative feedback [26] [27]. Terminal events (faints, evictions) impose explicit penalties [28]. The reward function and all coefficients are configured via YAML, and the system must fail-fast (refuse to run) if a configuration would violate design rules – e.g. a sanity check ensures the punctuality bonus is below the hunger-need weight so that skipping meals can never be rewarded [29].

The training loop integration is built in: Townlet exposes a PettingZoo-compatible multi-agent environment, so an external RL algorithm (e.g. PPO) can step the simulation and get batched observations, reward, and done flags [30]. The environment should handle *per-agent episode termination* (each agent can have its own done flag on exit while others continue) [31]. **Stability monitoring** is required to support training at scale – the system tracks metrics like lateness rates, starvation incidents, option-switch thrashing, etc., over rolling windows [32]. If a candidate policy causes a sharp regression (e.g. sudden spike in starvation or lateness), the system should flag it or trigger a rollback to a previous policy [33]. A separation between "shadow" policies (under training) and a "release" policy (serving the live simulation) is maintained, with promotion only when the new policy proves its stability over two evaluation windows [34] [35]. The simulation must support **live operations** as well: a human operator can issue console commands to spawn or teleport agents, adjust needs, toggle perturbations (random events like outages or price spikes), etc., with appropriate authentication and role control [36] [37]. All console operations and critical events should be logged with timestamps and identifiers (audit log), and a **telemetry publisher** should stream out state diffs, events, and metrics in real-time (e.g. via WebSocket or file) for the observer UI [38] [39]. In summary, Townlet's functional spec covers everything from low-level sim physics (grid, affordances, queues) up through RL integration, configuration management, and operator controls, to ensure the world behaves consistently with the design documents.

Non-Functional Requirements

Several non-functional requirements guide Townlet's design. **Performance** is a major factor: the target is to run at least *4,000 environment steps per second* with 8 agents × 8 parallel environments on baseline hardware [40]. The code should be optimized for vectorized operations and allow future acceleration (e.g. using Numba/Cython for hot paths or offloading pathfinding to native code) [41]. The simulation loop's real-time tick (250 ms) also implies a soft real-time requirement for the single-thread tick processing – any tick's processing should ideally complete before the next tick starts, to avoid drift (profiling of tick duration is planned to ensure new features don't add >5% overhead) [42] [43]. **Determinism** is explicitly required for reproducibility: the design mandates using three separate RNG streams (world events, perturbations, and policy decisions) whose states are captured in each snapshot [8]. This allows exact replay of simulations given the same snapshot and also helps isolate randomness sources (e.g. ensuring a change in policy RNG does not affect world event randomness). The system's **scalability** (in terms of agent count or map size) is considered in design – e.g. a spatial indexing structure is expected to keep neighborhood queries O(1) on average [44] [45], and embedding ID recycling is handled carefully so up to 64 agents can be tracked without ID collisions [46].

Stability and safety of learning are non-functional goals enforced via guardrails: reward clipping, termination penalties, and feature-flagged rollout of new mechanics ensure the training process remains stable. For example, even as new social rewards or perturbations are introduced, they are gated by config flags and A/B tests so as not to destabilise the running policy [47] [48]. **Data integrity** is important – any time the simulation's config changes in a way that would alter behavior, the `config_id` must be bumped and recorded (telemetry, snapshots) to avoid mix-ups [49] [47]. **Security & privacy** considerations are relatively light (since this is not a production service), but the design includes a "viewer" vs "admin" console mode to restrict destructive commands for general observers [50], and a privacy mode that can hash agent IDs in telemetry for public streams [39].

Another non-functional aspect is **maintainability and transparency**. The documentation is considered part of the deliverable – there is a full documentation plan to keep design docs updated in lockstep with implementation [51] [52]. The codebase uses Pydantic config models to validate inputs against design contracts (providing clear error messages if a YAML scenario violates allowed ranges or rules) [53] [29]. Extensive testing is required: the requirements outline golden tests for config invariants (affordance YAML schema, economic balance), property-based tests for things like economy or queue fairness, long-run fuzz tests (random teleports, outages mid-affordance to ensure no deadlocks) [54] [55], and even chaos engineering scenarios (removing agents or simulating telemetry drops to verify graceful degradation) [55]. Every milestone also carries explicit **acceptance criteria** – for instance, it's a requirement that by Milestone M2, no agent ever starves (need never drops to 0) and lateness stays under a threshold, otherwise the milestone isn't met [56]. These quality gates tie the non-functional aims (like stability, correctness, and performance) directly into the development and verification process.

Architecture Decisions

The Townlet design was shaped by key architecture decisions documented across the design files. One fundamental decision is to use a **modular, decoupled architecture** with well-defined interfaces between subsystems. The core simulation loop calls out to components (World, Queue Manager, Lifecycle, Policy, etc.) in sequence [57], and each subsystem exposes hooks or APIs so that future features can be added without breaking the contracts [58]. For example, the **affordance system is data-driven** – instead of hardcoding interactions, all affordances (actions agents can perform on objects) are defined in external YAML files loaded at startup [59]. This decision makes it easy to extend or modify game mechanics by editing config rather than code, and also ensures the sim enforces a strict schema for these definitions (fail-fast on invalid or duplicate affordance specs [60]). The architecture also emphasizes **hot-reloadability**: certain configs like affordances and events can be reloaded on the fly (with checksum validation) without restarting the simulation [61]. Other changes that would fundamentally alter dynamics (e.g. reward weights, network architecture) are cordoned off such that they require a restart, preventing inadvertent mid-run changes [61].

Another design choice was to build Townlet as a **headless simulation with a client-server style interface** for visualization and control. The simulation runs as a PettingZoo Parallel environment (Python process), and external components like the Observer UI or any future 3D renderer connect via a published API [62]. The interface is a JSON-based pub/sub telemetry stream (initial full state then incremental diffs) [63]. This means the sim can run on one machine (or many instances in training shards), while a web-based or native UI can attach to visualize state in real-time without being in-process. It was decided that any heavy-duty rendering or pathfinding logic *will not* live in the core Python loop – if advanced pathfinding is needed, an optional Rust/C++ microservice could be integrated via a stable FFI boundary [64]. This is a forward-looking choice to keep the Python loop lean and maintain 250 ms tick speed even as complexity grows.

Reinforcement learning integration was considered from the outset. The use of the PettingZoo API (ParallelEnv) was chosen to interface cleanly with existing RL libraries like RLlib or CleanRL for PPO training [30]. The environment provides standard `env.step()` and `env.observe()` methods per PettingZoo, and handles things like parallel agent termination. The team established guardrails for RL training – e.g. implementing a *release/shadow* policy system where the training (shadow) policy can be tested against metrics before being promoted to live (release) [34] [35]. This dual-policy design is an architectural decision to ensure continuous training does not destabilize the live demo. Similarly, the **use of feature flags for staged rollout** is a deliberate design: virtually every major system (relationships, social rewards, perturbations, etc.) has an off switch or stage gating in the config [65]. This allows enabling new complexity gradually (A→B→C phases) and supports A/B experiments (for example, deploying a new observation variant or reward scheme only on some runs until proven). The requirement that any config change should bump a `config_id` is an architectural policy to avoid subtle version drift – the code will treat mismatched config IDs as incompatible and require explicit migration steps [47]. This was implemented with a **snapshot migration framework** that can transform old saved states to new config versions if needed [66] [67], ensuring forward compatibility.

Finally, the architecture heavily prioritizes **observability and debugging support**. Telemetry publishing is not an afterthought; it's built into the core loop. Every tick, the TelemetryPublisher collects diffs and metrics and can output them to multiple sinks (stdout, file, or a TCP socket) with backpressure control [38] [68]. The system has a notion of schema versioning for telemetry so that the UI and other consumers can adapt to changes [69]. There's also an **audit log** for console commands and critical events (with idempotency tokens for safe retry) [70] [71]. All these decisions – modularity, config-drivenness, externalizing UI, RL integration, feature gating, and rich telemetry – were made to ensure Townlet is robust, extensible, and manageable as a platform and not just a one-off simulation.

Component Specifications

The high-level design defines the major components and their responsibilities clearly [72]. Below is a breakdown of Townlet's core components:

- **Core Simulation Loop** – Oversees the master tick cycle (250 ms per tick) and orchestrates all subsystems each tick [73]. It handles scheduling of operations in the correct order (console commands, events, agent actions, etc.) and maintains global timing (including any time dilation factor). It also manages three RNG streams (world, event, policy) for deterministic behavior and profiling/timing of tick execution [73].

- **World Model** – Represents the environment's state: a grid (default 48×48) of tiles and objects, agent placements, weather (e.g. sunny/rainy) and utility states (power/water on/off), and the economy variables (item stocks, prices, wages, rent) [74]. The world model loads interactive object definitions and affordances (from YAML) and provides queries for neighbor search. It is essentially the authoritative store of mutable state each tick.

- **Queue & Reservation Manager** – Manages queues at interactive objects and tile reservations to prevent conflicts [44]. Each interactive object can have multiple queue node positions; agents join queues and the manager enforces order and fairness (e.g. applying a short cooldown to agents who just left a queue to prevent immediate re-queuing). Reservations are used so agents "lock" a target tile/object during an action, with timeouts to avoid deadlock. Tie-break rules on simultaneous arrivals and a *ghost-step* mechanism resolve deadlocks [44].

- **Perturbation Scheduler** – Triggers random or scripted events in the simulation (utility outages, price spikes, “arranged meeting” events, etc.) according to configured probabilities and schedules [75]. It ensures fairness in how events hit agents (cooldown buckets per agent) and avoids overlapping too many events (staggering them if needed). The scheduler can also cancel or inject events via console commands (admin only).

- **Lifecycle Manager** – Monitors agents’ well-being and handles agent exits (death, eviction from town due to unemployment, “chronic neglect”, etc.) [76]. It enforces exit caps (e.g. at most N agents can be removed per day) and cooldowns between exits, to keep population turnover realistic. When an agent exits, this component queues up a replacement spawn after a delay. It also has a training-mode switch: in *training mode* (used in evaluation or live phases), certain failure conditions may be suppressed or handled differently [76].

- **Observation Builder** – Constructs the observation tensors for agents each tick in the required variant (full/hybrid/compact) [77]. It encodes the agent’s own state, a local map or feature vector, and social/personal context (like a snippet of top friendship/rivalry scores) depending on the variant. It also sets special flags like `ctx_reset_flag` to signal when an agent’s LSTM should be reset (e.g. on agent respawn or teleport) [77]. The observation builder is variant-aware and tightly linked to the `config_id` (changing the observation schema requires a config update and gating) [77].

- **Policy Runner (RL)** – Wraps the reinforcement learning policy (e.g. a PPO agent) and mediates between the simulation and the neural network policy [78]. It uses PettingZoo’s ParallelEnv interface under the hood. The Policy Runner handles action masking (so the policy can’t choose invalid actions), enforces an *option commitment window* (agents must stick with a high-level option for a minimum number of ticks) [78], and can incorporate scripted policies or behavior controllers for primitives. It also manages the annealing schedule when mixing behaviour-cloned policies (e.g. gradually transitioning from scripted to learned actions).

- **Reward Engine** – Calculates per-agent rewards each tick according to the configured reward model [79]. It sums components such as survival reward, needs penalties, wage accrual, punctuality bonus, and social rewards (when enabled) [80] [81]. It then applies guardrails: e.g. no positive reward if the agent has been marked for termination recently, clipping of reward magnitude per tick and per episode [26]. The Reward Engine also produces a detailed breakdown of reward components for logging/telemetry [82]. (This breakdown is exposed via telemetry to help operators see how each reward is composed.)

- **Stability Monitor & Promotion Manager** – Continuously monitors various metrics over time windows to assess policy performance and stability [83]. It tracks things like average reward variance, option-switch frequency, starvation or lateness rates, etc., using rolling 24h (sim-time) windows [32]. If metrics go out of bounds (canary triggers), it can flag the current training policy as regressing. The Promotion Manager side of this handles the logic of when a new policy can be promoted to “release”: for promotion, the candidate must meet KPI thresholds on two consecutive evaluation windows [35]. If issues are detected, it can hold back or trigger a rollback to a previous policy. This component ensures the live system only adopts new behaviors when they’re proven stable.

- **Persistence & Config Service** – Manages saving and loading of simulation **snapshots** and provides the interface for config feature flags [84]. Snapshots capture the entire world state, RNG states, and identity info (policy hash, config_id, etc.) so runs can be paused or transferred deterministically [85] [86]. The config service side exposes the current `config_id` and feature flag settings at runtime for use by other components (for instance, to check if a system is enabled). It also handles config migrations: when loading a snapshot with an older config_id, it checks if a migration handler is available and applies it or errors out [87] [67].

- **Telemetry & UI Gateway** – Streams simulation data to external observers [88]. It publishes an initial full state snapshot and then tick-by-tick diffs including agent states, events, economy changes, weather updates, KPI metrics, etc. This gateway implements backpressure (if the UI can't keep up, it will batch or drop frames) and supports privacy mode (e.g. hashing personal IDs) [89]. It also packages special payloads for the UI, like "policy inspector" info (e.g. policy network outputs or hidden state for debugging) and other debug tooling data (social graphs, event timelines, etc.) [88].

- **Console & Auth** – Exposes a control interface (via CLI or API) for humans to interact with the sim [90]. It requires authentication tokens (bearer tokens) for access, with roles distinguishing *viewer* versus *admin* modes [90]. The console processes commands like spawning agents, forcing perturbations, adjusting prices, toggling features, or approving lifecycle exits. It enforces an **idempotency token** on any destructive command (`cmd_id`) to ensure if the same command is received twice it isn't applied twice [90]. All console commands and their results are logged (with timestamps and issuer info) for audit. The console component also includes validation of inputs and provides helpful error codes (e.g. E_INVALID_ARG) for improper commands [37].

(Beyond these, there are subsystems for things like Embedding Allocation (assigning unique embedding slots to agent IDs and recycling them after a cooldown) [91] and Perturbation Scheduler (already noted), but they can be considered parts of the above major modules. The "PolicyRunner" also includes a behaviour cloning annealer and other training-specific helpers which become active in later phases.)

Integration & Dependencies

Townlet's design deliberately separates the core simulation from external dependencies, but there are key integration points. For reinforcement learning, Townlet is built to integrate with **PettingZoo** environments and by extension with RLlib, Stable Baselines, or other libraries that support PettingZoo's API. The simulation provides a `ParallelEnv` interface, making it straightforward to plug into a PettingZoo-compatible PPO implementation [30]. The **neural network policy** itself (the PPO model) is not part of Townlet's repository – it's an external dependency – but Townlet defines how it interacts (observations in, masked actions out). Checkpointing of the policy is handled by storing a `policy_hash` in snapshots and telemetry so that external training scripts know which policy version was in use [85] [92].

Another integration is with the **viewer UI**. The design treats the UI as an external client that connects via the Telemetry API. The Telemetry/Observer interface is specified to be a versioned JSON schema with initial state and then diff updates [63]. This decoupling means Townlet doesn't have a graphical interface built-in; instead, it outputs the data needed for one. The UI can be swapped or upgraded independently as long as it adheres to the telemetry schema. In practice, an ops dashboard or web app subscribes to Townlet's feed to render agent positions, needs, and events in real time. The **Observer API** is treated as a first-class interface, with version negotiation if needed [69] and documented payload examples in the repo (e.g. sample JSON logs) for integration testing [93].

The design anticipates needing more heavy-duty compute for certain tasks, so it allows **optional native extensions**. Notably, pathfinding (if the grid or agent count grows large) could be handled by a dedicated Rust/C++ module. Townlet sketches out a stable C ABI so that such a module could be integrated without disrupting the Python logic [64]. The idea is that you can plug in a compiled library for pathfinding behind a feature flag – if present, the simulation would call it for movement/path queries; if not, a simpler Python implementation suffices. This is not yet implemented, but the architecture notes it as a future scalability path [64]. Similarly, the simulation loop leaves room for replacing certain data structures with more optimized versions (e.g. using numpy arrays or even GPU tensors for state) as needed, again controlled via config flags or isolated in modules (the **acceleration plan** mentions possibly using Numba/Cython on hot spots later [41]).

Townlet's **external dependencies** are relatively minimal: aside from standard Python scientific stack (NumPy, etc.) and PettingZoo, the simulation doesn't rely on databases or external services. All state is in-memory, with snapshots saved to disk as needed. One dependency is PyYAML (for loading YAML configs), and Pydantic for schema enforcement – these ensure that integration with config files is robust. The simulation also includes connectors for things like sending telemetry to file or network; for instance, a simple file-based telemetry sink is implemented for demo purposes [94]. Security-wise, the only integration is with the environment for tokens (the console can read tokens from env vars for auth), and a simple secrets library is used to compare tokens safely [95] [96].

One can view Townlet as comprising three layers: the **core sim engine**, the **training harness**, and the **ops interface**. The core engine is self-contained (no external calls during run, deterministic). The training harness integration (RL libraries, etc.) happens at the edges – e.g. feeding observations to PPO, receiving actions. The ops interface (console/UI) is likewise decoupled via telemetry and command APIs. This layered design means Townlet can run in different modes: headless training shard (no UI, just producing log files), live demo mode (UI attached, console enabled), etc., without changing its fundamental behavior [62]. In a deployment scenario, you might run multiple **training shards** in parallel for faster experience gathering, each sending metrics to a central place, and one **release server** running the current best policy with the UI for viewers [62]. The documentation explicitly describes such a topology: multiple headless training instances (shadow policies) and a separate release instance serving the UI, plus a console service and possibly an external pathfinder service if needed [62]. All these integrate through the defined interfaces, underscoring the flexibility built into Townlet's design.

Data Models & Acceptance Criteria

Townlet's data models describe the structure of the world and agents. At the highest level, the **SimulationConfig** is the root of all configuration – capturing everything from feature flags to numeric parameters – and each running simulation is associated with a `config_id` that identifies its configuration version [97]. Within the world, the primary data structure is the **WorldState**, which contains: a grid of tiles (each tile has a type like floor, wall, road, etc.) and placed objects (each object has a type like bed, fridge, shower, etc. and is located on specific tile(s)) [98]; a registry of agents (each agent has an **AgentSnapshot** of dynamic state); and global states like **weather** (which can be sunny, rainy, cold – affecting agent comfort or prices) and **utilities** (booleans for power_on, water_on) [16]. The **economy** is represented by prices and stock levels for items (e.g. food price, inventory in store) and economic rules like daily rent and base wage rates [16]. These are configured such that one "money" unit roughly equates to the cost of a basic meal, and wages are set so that a full day's work covers a day's expenses with some slack [99] [21] (ensuring the economic loop is viable). The economy model also includes periodic restocks (e.g. store restocks daily) and inflation perturbations via the event system.

Each **Agent** in the world has several key data points: - **Needs**: hunger, hygiene, energy – continuous values in [0,1] that decrease over time and are increased by certain actions (e.g. eating raises hunger, showering raises hygiene) [100]. Needs have decay rates (configurable per need) and quadratic penalty in reward when low. - **Resources**: notably *wallet money*, which is separate from needs. Money is earned via wages and spent on food, rent, etc. It's not part of the homeostatic loss but indirectly motivates agents through needing to buy things [101] [102]. - **Job status**: whether the agent has a job, which job (e.g. "grocer", "barista"), and their current shift info (start time, end time, location) [103] [104]. Agent employment is scheduled via the jobs config – each job has a home location and shift times. The world tracks lateness or absence for shifts as part of the agent's context (cumulative late ticks, absent count) [105] [106]. - **Attributes/Personality**: each agent gets a set of fixed trait values on spawn, such as extroversion, forgiveness, ambition (in [−1,1]) [107]. These influence behavior subtly: extroversion can increase how much an agent seeks social interaction, forgiveness reduces how strongly they retaliate to negative events, ambition increases rivalry from work conflicts, etc. [108]. (Some traits like "patience" or "frugal" are mentioned in design but kept behind flags for potential future use [108].) - **Relationships**: if enabled, agents maintain a **ledger of ties** with other agents, with scores for *trust*, *familiarity*, and *rivalry* in each relationship [109]. These scores update based on events (sharing a meal increases trust, queue conflicts increase rivalry, etc.) [110]. To keep memory manageable, each agent only keeps the top-K strongest ties (e.g. 3 friends and 3 rivals) and forgets weaker ones when the ledger is full [111]. Relationship values naturally decay over time toward 0 if not reinforced. This data model is only active when `features.stages.relationships` is at least "B" – otherwise the simulation can ignore social ties beyond the rivalry needed for queues.

Data related to **queues** is also stored in WorldState: interactive objects have a list of positions designated as queue nodes, and the Queue Manager holds the current queue of agent IDs for each object (and who is currently using it). It also tracks any reservation locks. Additionally, a **rivalry ledger** exists (even if full relationship system is off) to record recent queue conflicts between agents – this is used to drive avoidance behavior and for telemetry (the conflict metrics) [112].

Regarding **acceptance criteria**, the project has documented clear definitions of done for each work package and overall testing goals. For each feature implemented, tests and metrics are defined to verify it meets requirements. For example, Work Package WP-01 (Affordance Manifest Compliance) is only "done" when the loader strictly validates all affordance YAML files, logs checksums, and tests cover missing/invalid entries [113]. WP-03 (Reward Guardrail Enforcement) is accepted when reward clipping, death-related suppression, and all bonus terms are correctly applied per the requirements doc, and unit tests verify each guardrail (no positive reward after death, etc.) [114]. The **Requirements** document itself has a section "Testing & Acceptance" that outlines tests like: running a 10k-tick fuzz simulation with random events and asserting no deadlocks or NaNs occur [115]; specific scenario tests (e.g. an agent in a blackout while showering should fail gracefully, no stuck reservations) [115]; KPI-based unit tests (e.g. a "lateness spike" event must trigger a rollback alert, or a config where starvation is possible must be caught by a test) [116]. There is also an emphasis on **A/B testing** for changes: for instance, if a new observation variant is introduced, an A/B run with the old variant vs new should show that key outcomes remain within 10% before it's accepted [55].

In summary, the data model is rich but well-structured: agents, world, and config each encapsulate distinct aspects of state, and numerous acceptance checks (unit tests, integration tests, and milestone exit criteria) ensure the implemented model aligns with the documented intent. By the final milestone, all acceptance criteria – including meeting the performance target and demonstrating the success metrics (no starvation, social ties formed, etc.) – must be verified as met [2] [117].

Documentation Gaps & Inconsistencies

Most of Townlet's design documents present a consistent picture, but a few gaps and minor inconsistencies were identified. First, the **Product Charter and Conceptual Design** appear fully aligned on vision and scope, but the planning documents mention a "Milestone Roadmap" which was not found in the repository. The documentation plan indicates a *Milestone Roadmap (M1–M9)* was to be produced in Week 1 of the recovery project [118], and the project plan refers to it for future scope [119], but no file named `MILESTONE_ROADMAP.md` is present. It's likely this was in progress or merged into the `PROJECT_PLAN.md`, but currently it's a missing piece of documentation. The **Architecture & Interfaces Guide** is labeled v0.2 [120], whereas the Conceptual and High-Level designs are v1.3, suggesting the architecture doc might not have been updated to incorporate all latest changes. For example, the architecture guide's Observation section notes "social graph wiring TBD" [121] – acknowledging that relationship data isn't yet integrated – while the Conceptual Design (v1.3) clearly defines how relationships work and even provides a design for social observation snippets. This indicates the architecture doc was slightly behind: it describes the plan for relationships integration but hadn't been revised after the social features were specced out. Similarly, the architecture guide doesn't explicitly mention the attendance state machine or unemployment cycle from the employment design brief (added later), though those are covered in separate design notes. These are not contradictions per se, but show the documentation was being incrementally updated. The team took snapshots of key design docs for planning purposes (under `docs/program_management/snapshots/` as noted in the README [122]) to avoid such drift, but some newer design briefs (like the **Attendance & Wage Brief** [123] and the **Reward Model Reference** for LR2 [124]) exist outside of those snapshots. It would be prudent to consolidate these into the main design spec to ensure one source of truth.

Another minor inconsistency is in naming of config flags in documentation vs implementation. The Employment Loop brief refers to an `enforce_job_loop` flag under "behavior" [125], whereas the actual config uses an `employment.enforce_job_loop` field [126] – a trivial difference, but it could confuse readers. Additionally, the design mentions traits like *patience* and *frugal* being behind feature flags [108]; the code indeed has a flag (`features.relationship_modifiers`) that likely controls some personality effects, but this isn't explicitly documented outside the code. The **Milestone Progress** document (`MASTER_PLAN_PROGRESS.md`) provides a snapshot of progress but is essentially an aggregation of other docs, and doesn't conflict with them.

In terms of content gaps: the Documentation Plan outlines several documents (Ops Handbook, RL Experiment Playbook, Testing & Verification Plan, Data Privacy Policy, etc.) [127]. Some of these are present (e.g. an Ops Handbook and a Data Privacy Policy exist in `docs/ops/` and `docs/policy/` respectively), but others like the detailed Testing/Verification plan were not explicitly found (though a "M6_VERIFICATION_NOTE.md" is referenced). It appears documentation was *in progress* alongside code – a natural outcome of a project recovery effort. The important design intents (architecture, requirements, work packages) are captured, and no major contradictions were found between them. The few inconsistencies are generally timing issues (docs lagging behind or planned docs not yet written). Overall, the intended architecture described in the docs is cohesive: any ambiguities (like how exactly an admin command would trigger a promotion) are clarified by references (e.g. to the Ops handbook or console contract). As the project moves forward, ensuring the architecture guide (v0.2) is updated to v1.x to include the relationship and attendance updates, and finalizing the milestone roadmap, will close the remaining documentation gaps.

Configuration-to-Component Mapping

Townlet's configuration is organized in YAML files (e.g. under `configs/`) which define simulation scenarios. The config schema is enforced by Pydantic models in `src/townlet/config/`. Below is a mapping of key config fields and feature flags to their corresponding runtime components or behaviors:

Config Field / Flag	Associated Components & Behavior
`config_id` (e.g. `"v1.3.0"`)	**Config Identity** – Acts as a version tag for the simulation's config and design contracts. It is embedded in snapshots and telemetry for provenance [49]. On loading a snapshot, if the `config_id` doesn't match the current config, the **Snapshot Manager** will refuse to load unless a migration handler is provided [87] [67]. This ensures config changes (like new features or different balance) are explicit and gated.
`features.stages.relationships` (`OFF` / `A` / `B` / `C1` / `C2` / `C3`)	**Social/Relationship Systems Toggle** – Controls if and when the **relationship ledger** and associated social features are active. If set to `OFF`, agents do not form trust/familiarity ties and no social snippet is added to observations. In stages `A` or higher, the relationship system is enabled (by Phase B it's expected on). The **Observation Builder** will raise an error if a social snippet is requested while this flag is `OFF` [128]. When ON, the **WorldState** starts tracking relationship metrics and the observation includes top-friend/rival info.
`features.stages.social_rewards` (`OFF` / `C1` / `C2` / `C3`)	**Social Rewards Toggle** – Controls the **Reward Engine**'s inclusion of social interaction rewards. At `OFF`, agents get no reward for chats or conflict avoidance. At `C1`, successful chats produce a small bonus (and failed chats or negative interactions can produce penalties) [81] [129]. At `C2`, the reward engine also grants a tiny reward for avoiding conflicts (queue avoidance) [81] [130]. `C3` would allow further modifiers (like scaling chat reward by relationship quality). The RewardEngine checks this flag each tick to decide whether to compute chat/avoidance reward components [129] [131].

Config Field / Flag	Associated Components & Behavior
`features.systems.lifecycle` (`on` / `off`)	**Lifecycle Manager Toggle** – Master switch for the **LifecycleManager** (agent retirement system). If `lifecycle: "off"`, the sim will not automatically terminate agents for needs starvation, old age, unemployment, etc. All agents effectively become immortal (unless removed via console). If `on`, the LifecycleManager actively monitors and flags agents for exit according to thresholds. This flag is intended for testing or phases where you don't want agents dying. The LifecycleManager code checks this to decide whether to evaluate exit conditions [22] (and presumably does nothing if off).
`features.systems.observations` (`hybrid` / `full` / `compact`)	**Observation Variant Selection** – Determines which observation **encoder** is used by the ObservationBuilder. All agents will receive observations in that format. For example, `hybrid` (the default) provides a medium-detail local map + some directional info, `full` gives a larger egocentric grid with path channels, and `compact` gives no grid at all (only compact feature vector) [132] [133]. This flag must be consistent with the policy: changing it requires retraining or at least an A/B test, hence any change should be accompanied by a `config_id` bump [47]. The SimulationConfig validation will reject unsupported values here [134].
`features.training.curiosity` (`phase_A` / `off`)	**Curiosity Reward Toggle** – Enables a small intrinsic reward for exploration (e.g. an RND bonus) during early training. When set to `phase_A`, the idea is that in Phase A (initial training), an exploration bonus is added to encourage agents to explore the environment [135]. The bonus would decay by milestone M2 and be off for later phases [135]. If set to `off`, no curiosity reward is applied. (In current code, this may be handled externally by the training script or a stub in the reward function – the config exists to toggle it).

Config Field / Flag	Associated Components & Behavior
`features.console.mode` (`viewer`/`admin`)	**Console Access Mode** – Sets the default access level for console commands. In `viewer` mode, only non-destructive commands are permitted (read-only operations like retrieving telemetry snapshots, or safe spawns). `admin` mode allows all commands including those that alter state (spawning agents, forcing events, promotion/rollback commands, etc.) [37]. The **ConsoleAuthenticator** and command handlers use this to filter commands – e.g. a "reset_world" command might only run if mode is admin. Typically, the public UI would use viewer mode, whereas developers use admin mode with a secure token.
`behavior.enforce_job_loop` (`true`/`false`)	**Employment Cycle Enforcement** – If true, the sim enforces all rules of the job attendance system (lateness penalties, absence tracking, firing agents after too many absences). If false, those rules can be relaxed. This flag was introduced as part of hardening the employment loop [125] so that new attendance features could be toggled off if they caused issues. When `enforce_job_loop: true`, the **World** and **LifecycleManager** will apply absence penalties and eventually mark agents for unemployment if they exceed `max_absent_shifts` [136] [137]. If false, agents likely won't be fired for missing work (making the simulation more forgiving).
`rewards.*` (numerical fields under rewards)	**Reward Function Parameters** – These config entries (e.g. `needs_weights`, `wage_rate`, `punctuality_bonus`, `survival_tick`, penalties, etc.) map directly to calculations in the **Reward Engine**. For instance, `needs_weights.hunger = 1.0` means hunger deficit is weighted 1.0 in the quadratic penalty [138]; `wage_rate = 0.01` means each tick at work yields +0.01 reward [139]; `faint_penalty = -1.0` applies a -1 reward when an agent collapses [140]. The RewardEngine reads all these values each tick from `config.rewards` [141] [142]. The model validators ensure these fall in sane ranges (e.g. wage_rate 0–0.05) [143] and enforce rules like *punctuality_bonus < hunger weight* (to keep incentives aligned) [29].

Config Field / Flag	Associated Components & Behavior
`rewards.clip.*` (`clip_per_tick`, `clip_per_episode`, `no_positive_within_death_ticks`)	**Reward Guardrail Settings** – These configure the clipping and suppression behavior in the Reward Engine. `clip_per_tick = 0.2` means any tick's reward is capped to ±0.2 [144]; `clip_per_episode = 50` caps the cumulative reward an agent can accrue over an episode (to prevent runaway positives over long runs) [144]; `no_positive_within_death_ticks = 10` means from 10 ticks before a termination to 10 ticks after, an agent cannot receive positive reward [145] [102] (this prevents an agent getting a big positive right as it dies). These values are used in the RewardEngine's logic when adjusting final rewards [146] [147].
`economy.*` (e.g. `meal_cost`, `rent`, etc.)	**Economic Parameters** – These feed into the **WorldState** and agents' financial calculations. For example, `meal_cost: 0.4` means a meal item costs $0.4 in-game, which will be deducted from an agent's wallet when they buy food. `wage_income: 0.02` (often equal to or related to job wage_rate) might define how much money an agent earns per tick of work (or could be legacy if wage is handled via rewards). The `rent` or similar fields define daily rent expense an agent must pay (though not explicitly seen above, it's mentioned in design). These values ensure the **economy balance** (basket of goods vs wages) – e.g. wage_income and prices should satisfy the invariant that a day's work covers a day's needs with some slack [21]. The Stability Monitor may watch these for sanity (e.g. if a config change violated the basket coverage, that's a problem).

Config Field / Flag	Associated Components & Behavior
`jobs.<job_id>.*` (start_tick, end_tick, wage_rate, lateness_penalty, location)	**Job Schedules & Pay** – Each job defined in config (e.g. `grocer`, `barista`) corresponds to an employment slot in the simulation. These settings are used by the **World** (or an Employment subsystem) to assign agents shifts and compute wages. `start_tick` and `end_tick` define the working hours (ticks in the day) [104]; `location` is the coordinates of the workplace; `wage_rate` is how much the agent earns per tick on the job (could feed into both reward and wallet) [104]; `lateness_penalty` might be a factor used when an agent is late – e.g. the config above shows 0.1, 0.12 which likely deduct from attendance or pay if late [104]. These tie into the **Attendance system** – e.g. if late, apply one-time penalty and reduce wage accrual as per design [148]. The LifecycleManager also uses these to decide when to fire an agent (after absences).
`employment.max_absent_shifts` (and other employment fields like grace, cutoff)	**Attendance/Unemployment Parameters** – Under `employment:` config (as per design brief [149] [150]), you can configure how strict the job attendance is. For instance, `max_absent_shifts: 3` means if an agent misses 3 shifts in a 7-day window, the **LifecycleManager** will trigger an unemployment exit for that agent [137] [151]. Other parameters like `grace_ticks`, `absent_cutoff`, `absence_penalty` correspond to the state machine in the attendance brief: e.g. grace period after shift start, cutoff to mark absence, penalty amount for being absent, etc. [149] [136]. These values are consumed by World/Lifecycle when computing attendance state each tick and deciding whether to increment absence counters or fire the agent.

Config Field / Flag	Associated Components & Behavior
`behavior.*` (e.g. `hunger_threshold`, etc.)	**Agent Decision Thresholds** – These are used by any scripted or heuristic behaviors. For example, `hunger_threshold: 0.4` means an agent (or the default behavior controller) will consider itself "hungry" when hunger need falls below 0.4 [152]. If using a scripted policy (Phase A), the agent might decide to eat when hunger < 0.4. These thresholds feed into the default routines or into the *BehaviourController* which provides baseline actions before learning takes over. They do not directly affect the core sim, but guide agent choices in the absence of an RL policy or alongside it.
`queue_fairness.*` (`cooldown_ticks`, `ghost_step_after`, `age_priority_weight`)	**Queue Management Tuning** – These config knobs directly configure the **Queue/Reservation Manager**. `cooldown_ticks: 60` means after an agent leaves a queue (or is involved in a queue conflict), that agent cannot immediately rejoin the same queue for 60 ticks [153] [154] – enforcing fairness by preventing rapid queue-hopping. `ghost_step_after: 3` means if an agent fails to acquire its target after 3 retries (e.g. due to collisions), the ghost-step mechanism will trigger on the next attempt [155]. `age_priority_weight: 0.1` gives a small bonus to the queue position for each tick an agent has been waiting, to favour those waiting longer [155]. These values are used in the queue manager's logic every tick it processes queues, and they align with the design requirements for fairness (tie-break adjustments, etc.).

Config Field / Flag	Associated Components & Behavior
`embedding_allocator.*` (`cooldown_ticks`, `reuse_warning_threshold`, `max_slots`)	**Agent ID Embedding Settings** – These configure the **EmbeddingAllocator** which hands out embedding IDs to agents for use in observations (so that each agent has a stable index for neural network inputs). `max_slots: 64` sets the maximum number of distinct embeddings (thus max concurrent agents recognizable) [156]. If more agents spawn beyond that, old slots have to be reused. `cooldown_ticks: 2000` means once an agent dies, its embedding slot won't be reused for at least 2000 ticks (approx 2 days) [156]. This prevents the network from confusing a new agent with the personality of a very recently departed agent [46]. `reuse_warning_threshold: 0.05` could be a threshold for telemetry: e.g. if embedding reuse rate (fraction of slots reused too soon) exceeds 5%, flag a warning [157]. `log_forced_reuse: true` indicates the system will log a warning event if it *does* have to reuse an embedding slot before cooldown [91]. These parameters ensure the identity mapping in observations remains stable and are referenced by the EmbeddingAllocator each spawn/ termination.
`affordances.affordances_file` (path)	**Affordance Definitions** – Points to the YAML file (or files) that list all affordances (actions and their effects) to load [158]. By default, e.g. `configs/affordances/core.yaml` would be loaded. The **World** reads this on startup to register all possible actions. If this path is changed, a different set of interactions would be available. If the file is missing or has schema issues, the config loader will fail (enforced by WP-01) [113]. This is how new gameplay mechanics can be plugged in without code changes – just by editing the affordances YAML and updating the path.

Config Field / Flag	Associated Components & Behavior
`affordances.runtime.instrumentation` (`on` / `off`) and `hook_allowlist`	**Affordance Hooks & Instrumentation** – These flags control runtime extensions for affordances. If `instrumentation: "off"` (as in the demo config) [158], the simulation might run in a minimal mode without extra logging or custom hook processing for affordances. Setting it "on" could enable additional debug logging each time an affordance runs (for developers). The `hook_allowlist` is a list of Python modules/ functions permitted to be called as affordance hooks [159]. By default it might include the built-in `townlet.world.hooks.default` module. This is a security/stability measure: only specified hooks will be executed when affordances have `before/after` hooks defined in YAML. Thus, these config values tie into the **World/ affordance execution** – controlling whether custom events (like special animations or narration hooks) fire during actions.
`stability.*` (`affordance_fail_threshold`, `lateness_threshold`, etc.)	**Stability Monitor Thresholds** – These config entries set the trigger levels for stability alerts. For example, `affordance_fail_threshold: 5` might mean if 5 affordance failures (exceptions or precondition fails) occur in a short period, the StabilityMonitor will flag it [160]. `lateness_threshold: 3` could mean if lateness ratio exceeds 0.3 (30%) it triggers a "lateness_spike" alert [161] [162] (the config is likely scaled: 3 means 0.3 in percentage). These values are used by the **StabilityMonitor** which ingests telemetry and computes alerts for ops. If thresholds are crossed, it can set flags or log warnings (for example, to not promote a policy that causes too high a lateness).

Table: Config fields and their linked runtime components. This mapping shows how scenario configuration (YAML) drives the Townlet engine. All config values are validated on load (e.g. numeric ranges in Pydantic) and many are cross-checked against design rules (e.g. bonuses vs weights, required fields). Feature flags in particular act as on/off switches that guard code paths in various subsystems, ensuring that incomplete or experimental features can be disabled in production runs.

Core Runtime & World Model Audit

Simulation Loop & Time Model

Townlet operates on a fixed-step simulation loop. Each **tick** represents 0.25 seconds of real time (250 ms) and the default config has 1000 ticks equal one in-game day [8]. The SimulationLoop enforces this timing, and can apply time dilation (e.g. 10× faster) or pause as configured. Internally, the loop proceeds through a well-defined sequence of updates every tick (the "tick pipeline"):

1. **Console Command Application** – At the start of the tick, any pending console operations are applied to the world (e.g. spawning or moving an agent, toggling a feature flag) [163]. These commands come with validation and will produce a result or error that gets logged. By processing them first, the sim ensures external inputs are synchronized to the tick boundary (and idempotency tokens prevent double-applying) [90].

2. **Event Scheduling** – Next, the **PerturbationScheduler** processes scheduled events for this tick [163] [164]. This may introduce exogenous changes like a price spike, a power outage, or a random social event if their trigger conditions (e.g. time windows, probabilities) are met. The scheduler respects fairness rules – e.g. it won't schedule two big events at once and uses per-agent cooldown buckets so the same agent isn't always targeted [165]. Events that occur are recorded in the world state (and will be part of telemetry).

3. **Observation & Action Selection** – The simulation then gathers the state needed for agents to act. The **Observation Builder** produces each agent's observation tensor according to the chosen variant (hybrid/full/compact) [163]. It computes the agent's own needs, some encoding of nearby entities, and any additional info (like shift status, social context if enabled) [166] [128]. Using these observations, the **Policy Runner** (and underlying RL policy) is invoked to obtain actions for each agent [163] [164]. This is essentially the RL policy step: the policy network processes observations and outputs an action for each agent, along with action masks and any meta-data. Invalid actions are masked out (the Policy Runner ensures masks are applied according to the sim's rules) [164]. The result is a set of intended actions (e.g. Agent 1 chooses "Interact with fridge", Agent 2 chooses "Move north").

4. **Affordance Resolution & Movement** – The chosen actions are then passed to the **World** for execution. The **Queue/Reservation Manager** first intervenes if necessary: for any actions targeting objects, it checks if the object is free or if the agent must queue [167] [59]. Agents that need to queue are placed in the appropriate object queue (with ordering by arrival time and tie-break rules). If two agents reach the same object at the same tick, the manager will tie-break based on their `patience` trait (if enabled) or random jitter [19]. It also automatically reserves the target tile/object for the agent at the front of the queue to prevent others from barging in [168]. Next, each agent's action is executed: movement actions update the agent's coordinates, interact actions trigger the corresponding **Affordance** logic, etc. Affordance effects (like consuming an item, changing needs or money) are applied to the agent's state and world state [17]. Hooks associated with affordances fire – e.g. a "on_finish_shower" hook might log an event or trigger a narration (only if the hook is in the allowlist) [17] [158]. If an affordance precondition fails (say, power is out so shower can't be used), the action fails gracefully and the agent may try something else next tick. The queue manager also handles **ghost steps** here: if an agent has tried and failed to get to an object repeatedly due to crowding, after N failures (`ghost_step_after` config) it will allow the agent to pass through others for one tick (temporarily ignoring collisions) to break a deadlock [19]. When that happens, it emits a `queue_conflict`/`ghost_step` event for debugging [169]. By the end of this stage, all agents have either acted or, if queued, will act in a subsequent tick when they reach front of queue.

5. **Needs Update & Decay** – After actions, the simulation updates all agent needs for the tick. This includes natural **decay** of needs (each need value is reduced by its decay_rate * tick_duration) and boosts/deductions from any affordances executed. For example, if an agent ate a meal, their hunger need might increase (i.e. hunger deficit decreases) by +0.5, but all agents will then lose

some hunger due to the base metabolism for the tick [17] [20]. The net effect is applied and needs are clamped to [0,1]. This stage ensures that simply doing nothing will eventually starve an agent, encouraging action.

6. **Lifecycle Checks** – With needs updated, the **LifecycleManager** evaluates each agent to see if anyone should be removed from the simulation this tick [22]. It checks various conditions: Has the agent's hunger or other need been at 0 too long (starvation)? Have they been flagged by an event (e.g. severe injury or policy decision)? Are they unemployed for too long or reached the max absent shifts? Each condition corresponds to a "termination reason." If any trigger, the agent is marked for exit. The LifecycleManager then enforces the rules: e.g. it won't remove more than `exits_per_day` agents – if that cap is hit, further exits are postponed. Agents set to exit are effectively "done" from the RL perspective (will get a termination signal). The manager also schedules **spawn events** for replacements: when an agent exits, after a short delay (e.g. 5–20 ticks, configurable) a new agent will be spawned [23]. The new agent will get a fresh personality and will usually take over the job vacancy of the old one (to keep workforce constant), unless config says otherwise. This respawn logic ensures population stability. Lifecycle also fires events: for example, if an agent died of starvation, it might emit a `agent_exit` event with reason, or if an agent is about to be removed for unemployment, it emits an `employment_exit_pending` event (giving the ops a chance to intervene via console) [22] [170].

7. **Reward Calculation** – Next, the **Reward Engine** computes the reward for each agent for this tick [171] [172]. It takes the updated world state and looks at each agent's situation: it adds a small survival reward (e.g. +0.002) [80], subtracts a needs penalty equal to the sum of squared need deficits weighted by `needs_weights` [80] [173], adds a wage reward if the agent was working this tick (based on `wage_rate` and whether they were on shift at the job location) [139] [174], and a punctuality bonus if they started the shift on time [175]. If social rewards are enabled (features.stages.social_rewards ≥ C1), it also processes any **social events** from this tick: the WorldState collects events like successful chats or conflict avoidance during the affordance phase, and the Reward Engine consumes those [176] [131]. A chat success might yield +0.01 base plus additional reward scaled by the trust/familiarity between the agents [48] [177]; avoiding a conflict might yield +0.005 if C2 is on [81]. Conversely, a chat failure could incur a small penalty. All these components (survival, needs_penalty, social_bonus, social_penalty, wage, punctuality, etc.) are summed for a raw reward [178] [179]. Then the guardrails are applied: if an agent was terminated this tick or very recently, any positive reward in their total is nullified (no "last moment" reward) [146]. The total reward is then clipped to ±`clip_per_tick` [147]. The RewardEngine also maintains an episode cumulative reward for each agent; if adding this tick's reward would exceed `clip_per_episode`, it reduces it so the cap is enforced [180]. The final per-agent reward values are stored, and a detailed **breakdown** of components is prepared for telemetry [181]. (If an agent was marked done, the environment will report the final reward and then reset that agent's episode totals.)

8. **Telemetry Publishing & End-of-Tick** – With the world state updated and rewards computed, the final step of the tick is to publish telemetry and perform any end-of-tick bookkeeping. The **TelemetryPublisher** takes a diff of the world state (positions, needs, etc. changes from last tick) and collates all events that occurred this tick (console commands results, perturbations, lifecycle events like deaths or spawns, queue events, etc.) [69] [182]. It packages these into an update message along with the current tick number and any aggregated metrics (e.g. current number of agents, average reward this tick, etc.). This is sent out via the configured transport (to the UI or log). Telemetry also includes specialized snapshots: e.g. the latest **reward breakdown** for each agent is published so operators can see the composition of rewards [82], and the latest **stability alerts/metrics** are published so the dashboard can indicate if something like a

lateness spike happened [32] [157]. The **StabilityMonitor** updates its rolling window counters at this point – it ingests things like the reward variance over the last N ticks, or how many starvation events occurred, etc., and if thresholds are crossed it might set an alert flag that goes out in telemetry [32]. For example, if lateness for the day went above the config threshold, it would mark a `lateness_spike=true` in the stability metrics [161]. If using the training harness, this is also where the system would log any training-specific info (like PPO loss metrics if applicable) – though in Townlet those come from external training code writing to its own log. Finally, if **autosave snapshots** are enabled (via `snapshot.autosave.cadence_ticks` config), the loop will check if it's time to save a snapshot of the world. If yes (say every 1000 ticks), it will invoke the SnapshotManager to dump the world state to a file (including `config_id`, RNG states, etc.) [183] [184]. With telemetry sent and snapshot saved (if due), the tick ends.

This loop repeats continuously. Thanks to the deterministic design, if you start from the same snapshot and run the same number of ticks with the same random seeds, you should get identical outcomes each time. The use of separate RNG streams for world and policy ensures, for instance, that adding a new random perturbation won't alter the sequence of policy decisions (they draw from independent entropy sources) [185]. The end result is a cycle where every game loop iteration advances all agents and systems a little, and then outputs an observational slice for us (or an AI) to consume. The loop is fast – the target of 4k steps/sec implies the internal processing of a tick is highly optimized, possibly batching operations where possible (e.g. computing all observations in a vectorized way). The actual performance in the current Python prototype might be lower, but the architecture leaves room for improvement (like moving collision checks to C++ if needed). The tick pipeline described above aligns closely with the design's "happy path" sequence [163] [164], ensuring all subsystems update in a logical order.

World Model & Economy

The **World Model** in Townlet encapsulates the simulation's physical and economic environment. It's essentially the state of the "town." Spatially, the world is a 2D grid of fixed size (commonly 48×48 tiles as per design) [74]. Each tile has a type (floor, road, wall, etc.) defining whether agents can occupy it or if it serves a special purpose (e.g. a counter tile for queues). The world also manages a list of **objects** placed on the grid – such as beds, fridges, stoves in houses, or shelves and registers in shops [98]. Objects can be static (like a bed is always in the house) or spawned dynamically (the design doesn't emphasize dynamic object creation, mostly static layout). Objects are interactive: many have associated affordances (e.g. a fridge might allow an "eat_food" affordance if the agent has food).

The World model contains the **economy state** which covers item stocks and prices. For instance, it tracks how much food is available at the grocer, the price of each item, how much money each agent has, and global parameters like daily rent cost. Each day a **restock event** might replenish stores (the design notes a "daily restock truck") [16]. Prices can be varied by perturbation events (inflation spikes) but are bounded to keep the game balanced [99]. An invariant stated in the requirements is that an agent's wage should roughly cover the cost of living: the config is tuned so that daily wage ≥ daily expenses (3 meals, etc.) with ~10–20% slack [21]. The system likely validates this by computing a "basket" cost and comparing to wage – indeed a KPI or test is mentioned to ensure wages cover at least 110% of basket cost [186]. If an operator drastically changed prices or wages in config, that could break this balance, but the expectation is that scenarios adhere to it.

Jobs and the economy: The world spawns agents with jobs (e.g. out of 6 agents, 2 might be shopkeepers, 4 regular citizens). Each job (like "barista" at the café) has a schedule and location defined in config [104]. The World's **job scheduler** is responsible for checking the current tick and determining if a job shift is starting or ending. When a shift starts, any agent assigned to that job should ideally be at the workplace. If they are not, the scheduler via the LifecycleManager may mark them late or absent and withhold wages accordingly [148] [137]. Wages accrue to agents while they are actively on shift (the simulation either increases their wallet directly or tallies wages to add at shift end). The employment system defines rules for *lateness*: being late by more than a grace period triggers a one-time penalty and reduces that day's wage (config `lateness_penalty`) [148]. If an agent never arrives by the cutoff, they are marked absent – they get no wages for that shift and incur an absence record (which can eventually lead to firing) [187] [136]. All these computations happen in the World/Lifecycle domain each tick around shift boundaries.

Weather and utilities: The world model may include a simple weather state (sunny, rainy, cold) that can affect agent behavior or needs (e.g. maybe being cold drains energy faster, rain could make agents prefer indoor paths). Utilities (power_on, water_on) are booleans that can be toggled off by events like outages [188]. If power is off, certain affordances (using a stove, taking a shower) have a precondition fail ("no power") [17]. Those events resolve automatically after some ticks (or can be ended by console commands). This adds environmental variety without a lot of complexity.

Spatial index: To efficiently handle agent movement and perception, Townlet employs a spatial indexing strategy. The high-level design suggests either a uniform grid bucketing or a quadtree for neighbor queries [44] [45]. In practice, since the grid is not huge (48x48), a simple grid-based indexing (dividing the map into regions) could be used. The goal is O(1) expected time for local queries – e.g. finding who is in the 11×11 observation square around an agent. The architecture doc explicitly notes this as a requirement for scaling beyond 50 agents [45]. The current implementation likely uses simple grid lookup (each tile knows which agent or object is on it), which is effectively constant time for neighbors within a fixed radius.

Agent state in World: Each agent has an **AgentSnapshot** in the world state containing their dynamic info. This includes: position (x,y tile), current action (if mid-affordance, e.g. "sleeping" with X ticks left), current needs values, current wallet money, whether they have a job and if they're currently on shift, and possibly things like a reference to their home. The simulation also tracks per-agent metrics like `lateness_counter` (how many ticks late today) [189], `on_shift` (bool if currently within a shift) [189], `attendance_ratio` (rolling ratio of on-time vs scheduled ticks) [105] [189], `wages_withheld` (money lost due to absences or lateness) [105]. These are updated by the employment logic. The agent snapshot may also include an `episode_tick` counter to know how long the agent has been alive (used for determining context resets and potentially value normalization) [190].

The world model is also responsible for **queue conflict tracking** and **rivalry**. When two agents conflict at a queue (one cuts in or blocks another), the system calls `WorldState._record_queue_conflict(agentA, agentB)` (as hinted by design) which will increment a rivalry score between those two [112]. Rivalry acts as a negative social tie that can cause agents to avoid each other in queues if it gets high. The Stability Monitor also monitors rivalry levels – extreme rivalry might indicate systemic unfairness. The **relationship system** builds on this: when fully enabled, the World maintains for each agent a map of ties (trust, fam, riv) to others. Trust and familiarity typically grow from positive interactions (sharing meals, successful chats) while rivalry grows from conflicts [110]. These values decay over time and the world will drop the weakest ties if an agent exceeds the max of 6 relationships stored [111]. Relationship updates happen in-world: e.g. when a "shared_meal" event occurs, the World might call a function to add +0.10 trust & +0.25 familiarity between the two agents [110]. These updates are likely queued as events which the simulation processes at end of tick (so as not to modify state mid-tick). The design also mentions that when an agent is removed, their relationships are cleared (and perhaps an event like "relationship ended" could be emitted, though not critical).

Economic transactions: The World handles agent wallets. Buying an item (affordance "buy_groceries") will deduct money from the agent's wallet and reduce stock in the store [16]. Getting paid (at shift end or per tick) will increase the wallet. Rent could be deducted daily at a certain tick (e.g. tick 900 of each day). These are straightforward updates to numeric fields in the world state but important for driving behavior (agents running low on money will seek work, etc.). Economic invariants, as mentioned, are expected to hold – likely there are assertions or at least tests to ensure no config accidentally makes, say, rent higher than max possible earnings (which would doom agents to eventual bankruptcy/starvation).

In summary, the World Model is like a sandbox of all simulation entities and rules: - It enforces physics (can't walk through walls, must queue for objects). - It enforces resource rules (can't take from an empty stock, can't spend money you don't have). - It coordinates subsystems like jobs, weather, etc., based on tick timing. - It is the single source of truth for anything that needs to be observed or saved (hence why snapshots capture the WorldState fully).

Queue & Reservation Systems

The **Queue and Reservation system** ensures orderly access to the town's limited resources. Many interactions in Townlet involve an agent using an object (e.g. a stove or a gym shower). To avoid two agents occupying the same object simultaneously, Townlet uses a reservation mechanism: when an agent is about to use an object, it **reserves** it (and perhaps the target tile) so no other agent can attempt to use it at the same time [168]. This reservation lasts until the affordance is completed or a timeout passes. If another agent arrives wanting the same object, they will see it's reserved and will typically join a **queue** for that object (if queuing is allowed for it). Each interactive object can define a set of queue positions (tiles adjacent or in a designated area). Agents queue up in an orderly fashion – essentially a FIFO unless priorities apply – occupying those queue tiles.

The Queue Manager implements a fairness policy to handle edge cases: - **Simultaneous arrival:** If two agents reach an object (or queue entry) at the exact same tick, there needs to be a deterministic way to decide who goes first. Townlet uses the `patience` trait when enabled: the agent with higher patience will wait, letting the lower patience agent go first, effectively (or it might be inversely – the design implies patience helps tie-break in favor of patient ones not cutting) [19]. If patience is not enabled (or equal), it falls back to a random tiny delay (jitter) to break the tie [19]. - **Queue position priority:** To prevent one agent from monopolizing an object by constantly re-queuing instantly, the config `cooldown_ticks` imposes that after using or leaving a queue, an agent must wait that many ticks before re-entering that same queue [153]. This stops rapid cut-ins. Additionally, Townlet adds an **age priority**: every tick an agent waits in line, it gains a small priority, so even if a faster agent arrives later, they won't always overtake a slow waiting agent if using something like "patience" (or even in absence, the slight weight makes the queue effectively FIFO with a tolerance for minor reordering) [153] [191]. This addresses starvation in queues. - **Deadlock prevention:** In a grid world, it's possible two agents block each other's way. The ghost step mechanism is designed to handle a form of deadlock: if an agent has failed to progress toward its target after N tries (the config `ghost_step_after`) [155], on the N+1-th try the sim will allow it to pass through obstacles for one tick [169]. Essentially, collisions are ignored for that agent, letting it move *as a ghost*. This breaks situations where agents keep swapping places endlessly. A **ghost_step event** is logged when this happens [169], since it indicates a serious conflict occurred. - **Reservation timeout:** If an agent reserves an object but doesn't actually use it (maybe due to some bug or because they got stuck), a timeout will free the reservation so others aren't blocked indefinitely. The timeout might be an implicit few ticks beyond the affordance duration or a config (e.g. an affordance fails if agent hasn't started it in X ticks after reserving).

While an agent is waiting in a queue, they effectively have an action “Wait” (or possibly “Queue”) and are not performing other affordances. Agents could abandon queues if their needs become critical – the design doesn’t specify, but that could be a future improvement (e.g. leave queue to eat if starving). The queue manager would then remove them from the queue list.

From a **component perspective**: the QueueManager is invoked during the affordance resolution phase of each tick. It likely has methods like `request_entry(agent, object)` which either grants the reservation if object free or puts the agent in queue. Each tick it might also advance queues: if an object becomes free (e.g. someone finished using it), it will allow the next queued agent to reserve it and start their action. The manager also tracks metrics: e.g. how long agents have been waiting (for telemetry), number of times ghost step was used, etc. [157]. These metrics feed the stability monitor (a high average queue wait or too many ghost steps might indicate a problem).

Telemetry and side-effects: The queue system generates events for noteworthy things. A `queue_conflict` event is emitted when two agents conflict at a queue (perhaps meaning one had to ghost step or one cut ahead) [112]. This event includes metadata (which object, which agents involved, maybe how long they waited, etc.) that is consumed by the telemetry publisher and possibly the narration system. Indeed, the design includes a **Conflict Narration** feature where, for example, if many queue shuffles happen, the narrator might output “Several citizens jostled in line at the café” – governed by throttling rules [192]. The queue system provides the raw data (conflict occurrences, queue lengths) that the narration system or stability monitor use.

Finally, the queue manager is tied to the **rivalry mechanic**: each time a “queue conflict” happens (like one agent cuts in front of another or they collide), the system increases the rivalry score between those two agents [112]. This makes them more likely to avoid each other in the future (the simulation might implement avoidance by, say, if rivalry > 0.7, an agent will choose a different queue or wait for the other to go first). Rivalry decay is handled in the relationships system over time [193]. All of this creates a feedback loop: repeated queue conflicts build rivalry, which in turn influences behavior to reduce future conflicts – a realistic social dynamic.

In summary, the queue & reservation subsystem ensures orderly resource usage and encodes fairness policies to avoid frustrating or unrealistic outcomes (like one agent hogging everything or indefinite stand-offs). It’s a critical piece for multi-agent interaction and has been designed with clear parameters (cooldowns, priorities) to satisfy the non-functional need for fairness. The code implementing this aligns with the design specs in docs (which call for queue cooldowns and ghost steps) [44] [18]. Testing likely included scenarios of multiple agents trying to use one resource to ensure queues form and ghost steps resolve deadlocks – crucial for proving this part of the simulation.

Tick Cycle State Changes & Outputs

During each tick, as described, the world undergoes a series of state changes. By the end of a tick, the following state updates typically have occurred: - Agents may have moved to new positions (if they had a movement action). - Some agents may have started or completed affordances, changing their state (e.g. an agent now has `is_sleeping=True` for the next 30 ticks, or an agent finished eating and their hunger is higher, an inventory item might be removed). - Needs values for every agent are decremented by decay and incremented or decremented by affordance effects. - Time-sensitive counters update: e.g. the world’s `tick` counter increases by 1; if `tick mod 1000 == 0`, a new day has started (which might trigger daily events like stock reset or rent deduction). - Job attendance info updates: if tick == shift start, agents not at work get marked late, etc.; if tick == shift end, agents leave work (the sim might automatically issue a move home or just mark them off shift and they’ll decide what to do next tick). - The `WorldState` might add new events to its event log: for example, if an agent became late, a `shift_late_start` event is recorded [194]; if an agent was absent at cutoff, a `shift_absent` event is logged [194]. If an agent reached an `on_time` state, maybe nothing is logged except internal counters. - If any perturbation events fired, global state toggles might have changed (e.g. power_on became False at tick 500 because a blackout event started, and a `power_outage_start` event is emitted).

All these changes are consolidated at tick end and fed into the TelemetryPublisher. The telemetry system then produces: - An **initial state snapshot** at tick 0 (for new clients or at sim start) containing the full world state. - Thereafter, **diffs each tick**. A diff includes all changes: positions of agents that moved, need values (often telemetry might send the full state of each agent each tick for simplicity, but it could optimize by sending only changed values), any events that occurred this tick (as a list), and updated metrics. - Some structured data like the “latest reward breakdown” per agent, which is a mapping of reward components from the RewardEngine [93]. - Aggregated metrics blocks: for instance, a “stability” block with any active alerts (e.g. lateness_spike: true) and current values of metrics like average option-switch rate [32]. - If any console command was executed, the result (success or error) is also included in telemetry (so that the UI can display acknowledgments). The console router likely keeps a “latest console results” entry [195] that is fed out.

One special type of output is the **narrative events**. The design introduced narration for conflicts and possibly other notable happenings. The telemetry includes a “narrations” or events stream for these as well [182] [196]. For example, if conflict narration is enabled and a conflict happened, the system might output a human-readable message (“A scuffle broke out at the café queue, but was quickly resolved.”). The **Conflict Narration Checklist** ensures these are throttled – config under `telemetry.narration` sets `global_cooldown_ticks: 30` (no more than one narration every 30 ticks per category) and `global_window_limit: 10` per hour [197] [198]. The TelemetryPublisher would enforce these, combining events if too many occur (e.g. summarizing multiple minor conflicts into one message) [192]. These narrated events are part of the tick outputs but they don’t affect the sim state – only the human observer experience.

Another output is related to **policy promotion/rollback**. While the actual training happens outside, Townlet’s StabilityMonitor and console support commands like `promote_policy` or `rollback_policy`. If a `promote_policy` console command is issued (and authorized), Townlet might update an internal pointer to load a new policy checkpoint file (though more likely, promotion just signals the ops to swap which policy the release instance is running – Townlet being a single process might not handle dynamic NN loading unless built in). However, Townlet does log promotions in a `promotion_history.jsonl` and sets a telemetry flag. The design says: when stability criteria met twice, raise promotion candidate; then ops triggers promotion, and the system logs it in `logs/promotion_history.jsonl` [199]. So as a side effect of a tick where a promotion happened, a log entry is written and telemetry might include a notice (e.g. `promotion_state: "promoted to vX at tick Y"` in stability metrics). Similarly, if a rollback occurs due to canary triggers, a log and telemetry flag would show that. These side effects ensure *traceability* of policy changes over time.

Finally, the **snapshot** side effect: if autosave is on, every N ticks a snapshot file is written (e.g. to `snapshots/` directory). This is a heavy output (contains entire world, RNG, etc.), and usually not done every tick to avoid overhead. The snapshot manager uses config like `autosave.cadence_ticks` and keeps only a certain number of recent snapshots (according to `retain`) [200]. Writing a snapshot does not alter the sim state (aside from maybe incrementing an internal snapshot ID); it’s purely an output.

To conclude, each tick transitions the world from state S(t) to S(t+1) through the pipeline of updates. The integrity of this transition is maintained by numerous checks (e.g. validation on config prevents illegal values that could cause NaNs, the code has assertions like hunger bonus < hunger penalty to ensure no contradictions). The side effects – telemetry, logging, events – provide a transparent window into these state changes for developers and operators. By auditing a few ticks during testing, one can verify that all these pieces are working: e.g. spawn a scenario where two agents want the same object and observe that one goes to queue, a queue_conflict event appears, rivalry increases, ghost step triggers on third try, etc., and that all corresponding config flags influence the behavior (turn patience off and see random tie-break instead, etc.). The current design and code, as per the documents, implement these systems in alignment with the specified requirements. The audit finds that the **core runtime loop and world model** operate as intended by the design: a deterministic tick-wise simulation with modular subsystems and rich telemetry, fulfilling the project's vision of an explainable, governable AI sandbox.

---

1 2 3 4 5 6 7 PRODUCT_CHARTER.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/PRODUCT_CHARTER.md

8 9 10 12 15 21 27 35 37 39 40 41 45 46 47 49 50 54 55 56 65 68 71 97 102 115 116 144 145 154 185 186 REQUIREMENTS.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/snapshots/REQUIREMENTS.md

11 132 133 189 190 OBSERVATION_TENSOR_SPEC.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/design/OBSERVATION_TENSOR_SPEC.md

13 14 16 17 18 19 20 24 25 30 31 33 48 98 99 100 101 107 108 109 110 111 135 153 168 169 188 CONCEPTUAL_DESIGN.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/CONCEPTUAL_DESIGN.md

22 23 32 34 36 57 58 59 69 89 91 112 120 121 157 163 164 165 167 170 ARCHITECTURE_INTERFACES.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/snapshots/ARCHITECTURE_INTERFACES.md

26 28 80 81 82 93 124 138 139 140 175 REWARD_MODEL.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/design/REWARD_MODEL.md

29 53 134 143 155 183 184 191 193 loader.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/config/loader.py

38 44 61 62 63 64 70 72 73 74 75 76 77 78 79 83 84 88 90 HIGH_LEVEL_DESIGN.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/HIGH_LEVEL_DESIGN.md

42 43 52 125 WORK_PACKAGE_EMPLOYMENT_LOOP.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/work_packages/WORK_PACKAGE_EMPLOYMENT_LOOP.md

51 118 127 DOCUMENTATION_PLAN.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/DOCUMENTATION_PLAN.md

60 113 114 WORK_PACKAGES.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/WORK_PACKAGES.md

66 67 85 86 87 92 200 WP05_DESIGN_NOTE.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/WP05_DESIGN_NOTE.md

94 104 126 152 156 158 159 160 197 poc_demo.yaml
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/configs/demo/poc_demo.yaml

95 96 auth.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/console/auth.py

103 105 106 123 136 137 148 149 150 151 161 162 187 194 EMPLOYMENT_ATTENDANCE_BRIEF.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/design/EMPLOYMENT_ATTENDANCE_BRIEF.md

117 119 199 PROJECT_PLAN.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/PROJECT_PLAN.md

122 README.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/README.md

128 166 builder.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/observations/builder.py

129 130 131 141 142 146 147 171 172 173 174 176 177 178 179 180 181 engine.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/rewards/engine.py

182 195 196 handlers.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/console/handlers.py

192 198 CONFLICT_NARRATION_CHECKLIST.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/design/CONFLICT_NARRATION_CHECKLIST.md