ChatGPT

Townlet Simulation Readiness Analysis

Simulation Flow and Core Loop

Townlet's simulation is driven by a main tick loop orchestrated in the `SimulationLoop` class [1]. Each simulation tick advances the world state through a fixed sequence of steps, closely following the project's high-level design. In the `SimulationLoop.step()` method, the engine performs the following in order [2]:

1. **Tick Increment & Respawns:** Increment the tick counter, update the global world tick, and process any pending agent respawns or removals via the `LifecycleManager` (e.g. replacing "dead" agents after a delay).
2. **Console Inputs:** Drain any queued console commands (admin interventions) from the telemetry and apply them to the world state, recording results for feedback.
3. **Environment Perturbations:** Advance the perturbation scheduler, which may inject random world events (like outages or price spikes) if their conditions are met this tick [2].
4. **Agent Decisions:** For each active agent, request an action from the policy layer (see next section) and then apply all agent actions to the world state [2]. This updates agent positions, object usage, queueing for resources, etc., according to the action types (move, interact, chat, etc.).
5. **Affordance Resolution:** Resolve ongoing affordances (multi-tick actions) and apply any end-of-day routines. For example, if the tick count hits the configured "day" length, the world enforces nightly resets or cleanup tasks (e.g. resetting daily counters) [3].
6. **Termination & Rewards:** Check termination conditions for agents (like hunger-based "fainting" or job exit events) via the lifecycle manager. Remove or mark agents that have exited, and compute rewards for each agent based on needs, social interactions, work status, etc., using the `RewardEngine` [4].
7. **Observations:** Build a fresh observation for each agent after the tick using the `ObservationBuilder`, encoding grid info and features (needs, positions, etc.). These observations are then passed to the policy module (or stored for training data) [5].
8. **Telemetry & Logging:** Publish a telemetry snapshot of the tick – including world state, agent observations, rewards, events, and health metrics – via the `TelemetryPublisher` [6]. This provides a detailed log or stream of simulation data for monitoring or UI. Finally, update stability metrics and promotion logic (used for evaluating when to promote a model or configuration) and prepare for the next tick [7].

This core loop is implemented and invoked by various run scripts. For example, the provided CLI script `run_simulation.py` simply loads a YAML config, instantiates the `SimulationLoop`, and runs a fixed number of ticks in a headless mode [8]. The simulation can thus run through ticks and update the world autonomously, albeit without a built-in visual display in this basic mode. The design is **event-driven and modular**, with each subsystem (world, agents, rewards, etc.) plugged into the loop via thin interface classes, making it easy to stub or replace parts for testing [9] [10]. Overall, the tick loop logic appears largely complete and follows the intended flow from initialisation to per-tick updates.

Agent–Environment Interaction

Agent behaviour in Townlet is handled by a policy module that integrates either scripted logic or RL policies for each agent. In the current implementation, a simple **scripted behaviour controller** is used as a placeholder for more advanced AI. During each tick, the simulation calls `PolicyRuntime.decide()` which iterates over all agents and produces one action per agent [11]. Agents not under any manual "possession" are controlled by the `BehaviorController` (scripted policy). This controller (implemented in `ScriptedBehavior`) evaluates the agent's situation and needs, deciding on an `AgentIntent` – e.g. move towards a job at start time, satisfy hunger by queuing for the fridge, take a shower if hygiene is low, initiate a chat if social conditions allow, or simply `wait` if nothing urgent [12] [13]. These intents are then converted to concrete action dicts (with fields like `kind`, `object`, `affordance`, target positions, etc.) for the environment to process [14].

All agent actions for the tick are collected and then applied to the world via `WorldState.apply_actions()` [15]. This method handles each action type atomically, updating the simulated world: for example, a `"move"` action directly changes the agent's grid position [16], a `"request"` action enqueues the agent for using a particular object (e.g. lining up to use a shower or stove) and may block if another agent is ahead [17], a `"start"` action will begin an affordance (e.g. start using an appliance), and `"chat"` will validate if two agents are co-located and then log a social interaction event. The code ensures these actions take effect on the corresponding agent's `AgentSnapshot` (which tracks state like needs, last action outcome, etc.) and on world objects (like marking an object occupied or updating queues). After all actions are applied, the world then calls `resolve_affordances()` to advance any ongoing multi-tick actions and free up resources that have completed [18].

Observations are generated after actions and outcomes are resolved. The `ObservationBuilder` constructs an observation for each agent comprising a local map (nearby agents/objects), feature vectors (e.g. normalized needs levels, time-of-day sin/cos, whether the agent is on shift at work, last action success, etc.), and social metrics [19] [20]. This gives each agent a structured view of the state for decision-making. In the current scaffold, these observations are primarily used for logging and for feeding into a training buffer if doing RL training – agents' next actions in the running sim are still decided by the scripted policy each tick (unless an RL policy network were plugged in later).

The scheduling of agents is effectively simultaneous and tick-based – each tick every agent gets one action opportunity. The simulation does not use a continuous time or asynchronous schedule; instead it advances in lockstep ticks, which simplifies fairness. The `PerturbationScheduler` adds environment-driven events in parallel, but agent actions are processed in a loop with no explicit priority – all actions in a tick use the world state from the start of that tick, and their effects accumulate for the next tick. There are mechanisms to avoid conflicts (e.g., a **queue manager** for shared resources ensures only one agent can start using an object at a time), and if an agent can't get an immediate resource it may have an action marked `"blocked"` which the logic will handle by trying again or dropping the intent [21] [22]. Overall, the agent–environment interaction loop is implemented to a functional degree: agents receive observations each tick and output actions, and the environment applies those actions affecting agent state and generating new observations. Complex interactions like social influence, job attendance, and needs decay appear to be modelled, although these systems might need tuning. The design is such that plugging in a learning policy later is feasible because the PettingZoo/PolicyRuntime scaffolding is already in place [23].

Output and Visualisation

At present, the simulation's outputs are primarily **telemetry and logging** rather than a polished graphical UI. Each tick, the `TelemetryPublisher` collects a rich snapshot of what happened and either logs it or sends it to an external dashboard. The telemetry includes per-tick records of observations, rewards, events, and health stats (e.g. tick duration, queue lengths) [6] [24]. By default, the engine writes JSONL logs for certain data (for example, promotion history, console commands) in a `logs/` directory, and prints basic info to console if running in verbose mode. This logging is useful for debugging or offline analysis. There is also an **admin console interface** integrated into telemetry: one can queue commands (like forcing a specific event or querying status) that the simulation will execute on tick boundaries, and the results are captured in telemetry outputs [25] [26]. The provided `console_dry_run.py` script demonstrates this by issuing commands (e.g. snapshotting state, reviewing employment exits) and printing the outcomes as a form of textual debug output [27] [28].

For visualisation, a separate **observer dashboard** is scaffolded. The repository includes `scripts/observer_ui.py`, which launches a Townlet observer dashboard for a running simulation [29] [30]. This script instantiates the simulation loop and then calls `townlet_ui.dashboard.run_dashboard(loop, ...)` to start an interactive UI. The UI likely displays the grid world and agent info in real-time, given parameters like refresh interval and focus agent [31] [30]. However, the `townlet_ui` module comes from an external or separate package (it's referenced but not contained in the core `townlet` directory), so the dashboard is an optional component. It suggests a Streamlit or similar interface may be used to render the 48×48 grid and agent statuses. Since the simulation core is headless, this dashboard subscribes to the telemetry or uses the simulation loop directly to fetch state each tick.

As of now, there is **no built-in game GUI** (no native graphics within the `townlet` core). The "UI modules" are present but noted as scaffolded (incomplete) [10]. Therefore, any demonstration would rely on either the external dashboard or examining the log outputs. In practice, one can run the simulation for N ticks and then inspect the JSON logs or use the console commands to get snapshots of agent states. The presence of a real-time dashboard script is promising, but its polish and capabilities depend on the external UI implementation. In summary, **output is mainly via data/log streams** (suitable for developers or integration with analysis tools), and a basic dashboard exists for visualization, but a fully featured in-game UI is not yet part of the core project.

Configuration and Integration Glue

Townlet is highly configurable: all scenario parameters (grid size, agent definitions, affordances, jobs, events, etc.) are loaded from YAML config files. The `configs/` directory contains example scenario configs (e.g. `base.yml`, `poc_hybrid.yaml`) which define the world setup and feature flags. The `load_config` function reads these and produces a `SimulationConfig` dataclass that centralises all settings. At runtime, this config is used to initialise every subsystem. For instance, `SimulationLoop(config)` triggers building of components like `WorldState.from_config` (which sets up data structures for agents, objects, and affordances based on config contents) [32] [33]. If the config includes predefined agents or objects, those would be created here. The code also seeds random number generators for different domains (world, events, policy) with seeds derived from the config ID to ensure reproducibility [34].

All the pieces are glued together in the simulation loop's constructor. It instantiates the world grid, the lifecycle manager (for respawn/exit logic), the perturbation scheduler (random events), the observation builder, the policy runtime, reward engine, telemetry publisher, stability monitor, etc., using factories or

default implementations [35] [36]. Notably, some components allow extension via config: e.g. the affordance execution backend can be swapped by specifying a factory in config (the code will import a custom class if provided) [37] [38]. This indicates the integration points are designed to be flexible.

In terms of launching the simulation, **multiple CLI entry points** are provided in the `scripts/` folder for different purposes [39]. We have already mentioned `run_simulation.py` for a straightforward headless run. There's also `run_training.py` (referenced in docs) intended to exercise the training harness with PPO/BC integration stubs [40]. More specialised scripts exist, such as `benchmark_tick.py` to measure tick performance [41], `capture_rollout.py` to run a simulation and save trajectories for offline training, and `run_mixed_soak.py` which alternates replay and live rollouts to stress test the training loop [42] [43]. These scripts illustrate how the core simulation can be embedded into different workflows. For interactive use or demos, the `observer_ui.py` as discussed ties the loop into a dashboard; for research, the PettingZoo environment interface is planned via the `PolicyRuntime` so that agents could be trained with RL algorithms by treating Townlet as an environment. The key integration "glue" is that once a config is loaded and `SimulationLoop` started, everything else (be it UI, training, or analysis) connects to it through the telemetry or policy APIs.

Overall, the config-driven design and the presence of these scripts mean the project has **multiple integration points** but not a singular "game launch" command yet. A user must pick the appropriate script depending on the goal (simulation demo, training run, etc.). The glue code is in place to load scenarios and run the loop, but a bit of technical know-how is needed to invoke the right components for a demonstration.

Implementation Status and Missing Pieces

The Townlet repository is clearly under active development, and while the foundations are solid, some features are still **marked as scaffolding or TODO**. According to the project README, the simulation, training, and UI modules are scaffolded with interfaces in place but not fully realised [10]. In practical terms, this means:

- **User Interface:** There is a basic observer dashboard hook, but no in-engine UI or polished visualization yet. The UI code exists as a scaffold (likely requiring the separate `townlet_ui` module). For a true demo, a more complete visualisation (even if 2D grid rendering) would need to be finalised. As it stands, the demo would be "headless" or developer-oriented, showing logs or a simple dashboard rather than a game-like interface. This is a notable gap for a general audience demonstration.
- **Agent Content and AI:** The agents currently follow a simple rule-based policy. This suffices for a proof-of-concept of emergent behavior, but the *reinforcement learning aspect is not yet operational*. The hooks for PPO training and blending scripted with learned policy are present (the code mentions annealing between scripted and learned actions), but training a model would require additional work and tuning. Until an AI policy is trained, the demo will rely on the scripted behaviours which, while functional, might be deterministic or limited in complexity.
- **Simulation Detail:** Some world subsystems might be stubbed or minimal. For example, the economy, relationships, and rivalries systems exist in code but their effects might be simplistic. The config allows enabling or disabling features ("stages"), suggesting some features can be turned off if incomplete. There are also likely placeholders for content: e.g. the affordance definitions (activities agents can do) are loaded from config, but if those configs are sparse, agents may have only a few basic actions. The good news is the code structure can support richer behavior; the work remaining is to populate and fine-tune those configurations (e.g. add more object types, more nuanced needs effects, etc.). No obvious outright *broken* pieces were

found in the code review – it's more about completeness and depth of features rather than fixing known bugs.

- **Testing and Stability:** Given the many components, a full integration test (all systems running together for long simulations) might not have been completely validated yet. The presence of a stability monitor and promotion manager indicates the team is aware of the need to detect when the simulation becomes unstable (perhaps agents all dying or metrics diverging). For a short demo (hundreds of ticks) this is probably fine, but longer runs or unusual configs could surface issues. In the repository, many issues are tracked on GitHub to implement remaining items, which implies some parts are awaiting completion (the README mentions work tracked via issues and milestones [10]).

Final Assessment and Demo Readiness

How close is Townlet to a working demonstration? From a software standpoint, the core engine is mostly in place: one can load a scenario config and run the simulation loop to produce emergent agent behaviours on a grid [8]. The crucial systems for a minimum viable demo – agent logic, environment updating, and logging – are implemented and appear to function coherently together. In that sense, Townlet could already demonstrate agents moving around, interacting with objects and each other, and accumulating rewards over time. The **concept** of the simulation (a small town with 6–10 agents in a 48×48 grid) is implemented at a scaffold level, and short runs have been smoke-tested (as evidenced by the existence of a smoke test in `pytest` and the example of running 100 ticks in the README quickstart [44]).

However, for a polished **public demo** (or even an internal demo to non-developers), there are still significant gaps. The lack of an integrated visualization means the demonstration would likely be textual or require an external tool. This reduces its impact, as viewers can't easily watch the agents in the grid world unless the separate dashboard is employed. Additionally, since the AI is currently scripted, the "emergent behaviour" is somewhat constrained to what's been hardcoded (though still interesting, it might not yet showcase learning or adaptation). To reach a true demo-ready state, the team would need to finish the UI/dashboard integration and perhaps have a scenario with clear, observable outcomes (for example, agents successfully maintaining their needs or going to work and getting fired for low performance, etc., over a short session).

In percentage terms, one might say the project is roughly **80% of the way to a basic demo**. The engine runs and produces data (which is the hardest part), but the final 20% – which includes user-facing elements like visualization, fine-tuning agent behaviour, and smoothing out configuration – is in progress. There do not appear to be major architectural pieces missing, just incremental feature completion. If pressed, a developer could demonstrate Townlet today by running the simulation and using the observer dashboard or console outputs to explain what's happening. Yet, to an observer, this would feel more like a development preview than a finished simulation game.

In summary, the Townlet simulation core is largely complete and correctly orchestrates a tick-by-tick virtual world with agents and events. Agents can act and react in the environment, and the system tracks all necessary information for complex behaviours to emerge. What remains is mostly **integration polish**: building a richer UI/visualisation, ensuring all planned features are toggled on and fleshed out, and perhaps integrating a trained RL policy to replace the current scripted policy for a more dynamic demonstration. With these pieces addressed, Townlet will be ready for a compelling demo of a miniature living world. As of now, it's a solid foundation that is on the cusp of being demo-ready, pending those final components and refinements. [10] [45]

---

1 2 3 4 5 6 7 9 15 24 34 35 36 37 38 sim_loop.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/core/sim_loop.py

8 run_simulation.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/scripts/run_simulation.py

10 44 README.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/README.md

11 14 runner.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/policy/runner.py

12 13 21 22 behavior.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/policy/behavior.py

16 17 18 32 33 grid.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/world/grid.py

19 20 builder.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/observations/builder.py

23 39 40 45 AGENTS.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/AGENTS.md

25 26 27 28 console_dry_run.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/scripts/console_dry_run.py

29 30 31 observer_ui.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/scripts/observer_ui.py

41 benchmark_tick.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/scripts/benchmark_tick.py

42 43 run_mixed_soak.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/scripts/run_mixed_soak.py