ChatGPT

Forward Work Program for Townlet Tech Demo

Tech Demo Readiness – Core Work Packages

- **Restore Simulation Loop Execution & Smoke Tests:** First, fix the CLI simulation runner so that it actually advances ticks. Currently, the `run_simulation.py` script instantiates the loop but never iterates it, meaning no ticks occur [1]. Implement iteration (e.g. looping over `SimulationLoop.run()`), then add a smoke test to verify ticks advance and telemetry is produced [2]. This ensures the flagship CLI works properly for demo runs.

- **Strengthen Core Loop Stability with Tests:** Add integration tests covering the simulation CLI and training CLI end-to-end to catch regressions [3]. For example, a test should run a few ticks via the CLI and confirm the world state updates (tick counter, agent state changes) [4] [5]. Additionally, provide a safer API for running the loop (e.g. a `run_for_ticks()` helper) to prevent misuse of the generator interface [6]. These measures guard against future breakage so the demo isn't derailed by basic errors.

- **Complete Observation & Reward Plumbing:** Finalize any "TODO" features in the observation and reward system that could affect the demo. In particular, implement the **compact observation variant** that's currently a stub [7] so that all configured observation modes work reliably. Ensure reward guardrails (e.g. clipping, termination conditions) are in place per design to keep the simulation stable during long runs [8]. With observation tensors and rewards solidified, the agents' behavior will be consistent and debuggable during the demo.

- **Optimize Performance for Longer Runs:** Improve the simulation loop efficiency to handle the 6–10 agents envisioned in the demo without slowdown [9]. For example, optimize the world-state "local view" and observation building, which currently recompute neighbor data for each agent every tick (On^2 scaling) [10]. Caching spatial indexes each tick (so agents can share the same world snapshot) will boost tick throughput [11]. Faster tick processing (aiming for thousands of steps/sec in headless mode) ensures that training and demo simulations run smoothly [12], and allows showcasing longer simulations or higher agent counts if needed.

- **Telemetry Reliability & Diff Streaming:** Harden the telemetry pipeline so we can confidently connect a live dashboard or GUI. Right now the telemetry publisher sends full world snapshots each tick, which is heavy [13]. Implement **diff-based telemetry** or channel filtering to send only changes or summary metrics each tick [14]. This will reduce bandwidth and GUI processing load, preventing delays or buffer overflows in the live feed [15]. Additionally, expose telemetry health metrics (queue lengths, drop counts, worker status) so we can detect if the dashboard is falling behind [16] [17]. By streaming lean, relevant telemetry (with proper buffering and retries), the demo viewers will get timely updates without stalling the simulation.

- **Finalize Observer Dashboard UI:** Complete the development of the Rich-based console dashboard that will let us **watch the simulation in real time**. The foundation is in place (various panels for agents, social ties, economy, etc.), but some features remain to be finished. Key tasks include implementing the **per-agent info cards** and any pending overlays. For example, add the proposed agent card grid with two columns listing each agent's vital stats (needs bars for hunger/hygiene/energy, current job/shift, wallet balance, rivalry status, recent social info) [18] [19]. Ensure the dashboard auto-updates every tick (e.g. ~1s refresh) and can handle the full breadth of metrics (KPI panel, perturbation banner, etc.). One deferred item is the **audio toggle** for the UI – decide on how to incorporate audio cues or alerts, implement that toggle (pending UX approval), and update documentation accordingly [20] [21]. By polishing the console UI, we'll have a rich, at-a-glance view of the town simulation for the tech demo.

- **Decide on Web GUI vs. Console:** Concurrently, evaluate if a web-based GUI is needed for broader accessibility. The current plan leans on a terminal dashboard (Rich/Textual), which is nearly complete, but a web UI could offer graphical visualization. We should weigh the effort and **decide on the UI toolkit** soon [22]. For the tech demo, the console UI with telemetry may suffice (especially given time constraints), but we will keep the telemetry interface open so a web dashboard can be attached. If resources allow, prototype a minimal web viewer that connects to the telemetry stream (e.g. a simple web app showing the grid and agent stats in a browser). Otherwise, proceed with the Rich console and ensure it's documented (so operators know how to run it and interpret it). Making this decision early prevents delay in UI polish and guarantees we have a reliable way to observe the demo.

- **Demo Scenario Scripting & QA:** Develop a compelling **demo scenario** with a scripted timeline of events to showcase emergent behavior. Using the built-in `DemoScheduler`, we can schedule key events at specific ticks to bring the simulation to life [23]. For example, start with agents spawned and employed, then at a certain tick introduce a **new agent arrival** (simulating a "new resident" joining – e.g., `guest_1` appears by tick 5 [24]). Later, trigger an **environment perturbation** like a price spike in the economy at tick 20, which should be reflected in the dashboard's perturbation banner [25]. We can also script a social event (e.g. force two agents to chat or cause a conflict) to demonstrate the relationship dynamics. All these scheduled actions should use the command palette or timeline YAML so they execute automatically during the demo. After scripting, perform dry-runs and follow the demo playbook to verify the expected visuals: the agent panels updating, the perturbation appearing, command palette logs of the scripted commands, etc. [25]. This workpackage ensures the **tech demo tells an interesting "story"** – we seed initial conditions (jobs, relationships) and then let the simulation plus a few triggered events produce an engaging narrative. We will capture logs, screenshots, and telemetry from rehearsal runs as artifacts to confirm everything is working before the live demo [26].

- **Integration of Learned Agent Policies:** To highlight the "DRL" aspect of Townlet (deep reinforcement learning agents in a Sims-like world), integrate the latest trained policies into the simulation. By now, **behavior cloning (BC) and PPO training pipelines have been developed** (Milestone M5 completed) [27]. We should use those to provide our agents with a reasonable baseline of behavior for the demo – for example, an RL policy that was trained to satisfy needs and go to work on time, derived from the scripted agents [28]. Work tasks here include: running the BC training to mimic scripted routines, conducting an annealing phase where agents gradually take over from scripts, and selecting a stable policy checkpoint for the demo. We'll test the chosen policy in the simulation to ensure agents indeed exhibit sensible daily routines (e.g. they seek food when hungry, use the shower, go to their job, socialize occasionally) without diverging into erratic behavior. If any issues arise (agents getting stuck or "collapsing" their needs), we may tweak rewards or fallback to more scripted guidance in certain areas. The end goal is to **showcase emergent routines** driven by an AI policy – fulfilling the vision of agents growing from simple scripts into believable autonomous behaviors [9]. This will demonstrate the viability of Townlet's RL sandbox for social storytelling.

- **Operational Preparations:** Finally, carry out the operational checks and polish needed for a smooth tech demo. This includes updating documentation (the demo playbook, runbooks) with any new commands or parameters, and making sure all config files for the demo (the base scenario config, timeline script, perturbation plan) are cleaned up and versioned. Implement any quick safeguards needed for live operation – e.g. ensure there's an easy way to pause or slow the simulation if a presenter needs to explain something, and a way to reset or exit cleanly. Since the demo will involve live operations (possibly using the command palette for perturbations or queries), double-check that the **console commands** (both viewer and admin mode) are secure and work as expected. If not already done, we might introduce a minimal access control for the console in demo settings (so that only authorized users trigger admin commands) – full authentication is likely overkill for a closed demo, but at least guard the dangerous commands behind an "admin mode" toggle that isn't exposed to general viewers [29] [30]. Confirm that telemetry and console output is logged to files as well (so we can later analyze or share the emergent stories that happened). With these operational items checked off, we'll be ready to execute the tech demo confidently.

Simulation & Agent Enhancements – Increasing Entertainment Value

- **Dynamic Perturbations & Story Events:** To make the simulation more entertaining and realistic, implement a variety of **perturbation events** that can occur in the town. This is aligned with the upcoming Milestone M8 which focuses on perturbations and narrative events [31]. Concretely, we'll extend the perturbation scheduler to support events like power outages, market changes, or other town incidents. For example, a **power outage** could temporarily disable certain affordances (as we partially have with showers losing power [32]), affecting agents' ability to satisfy needs and forcing them to adapt. A **price spike** or economic downturn event can change the cost of meals or wages, testing agents' budget management. We'll incorporate fairness logic so events impact agents even-handedly or according to scenario rules (e.g. a price spike hitting everyone's wallet proportionally) [33]. Each such event will produce telemetry/narrative output – for instance, a narrative log entry like "A thunderstorm knocks out power in town at tick 100!" – so the audience gets a story-like context. By scripting a few of these perturbations (or triggering them via the command palette during live play), we introduce unscripted challenges that agents must handle, leading to **emergent behaviors** (e.g. agents queue up for the only working shower, or form new social ties by "sharing" resources during a crisis). These narrative events make the simulation feel alive and increase viewer engagement.

- **Agent Personalities & Diversity:** Introduce configurable **personality profiles** for agents to create variability in behaviors. In the current system all agents follow the same rules/rewards; adding personality traits will make their decisions more varied and realistic (as planned for Milestone M9) [34]. For example, one agent might be **more social** (derives higher reward from chatting and tends to seek others more often), while another is **more work-focused** (prioritizes getting to work on time and saving money). Some could have higher or lower thresholds for needs (e.g. a "glutton" gets hungrier faster, a "neat" personality gets unhappy if hygiene drops even a little). We can implement this by tweaking need decay rates, reward weights, or action selection bias per agent via a personality config. We'll also add a few **pre-set personality archetypes** (like "Friendly", "Competitive", "Lazy", etc.) to easily configure different agents. This enhancement will lead to more **emergent interpersonal dynamics**: for instance, a highly social agent might initiate conversations that others wouldn't, causing friendship networks to form, whereas a competitive agent might hoard resources or rush for limited spots (sparking rivalries). In the demo and beyond, viewers will notice distinct behaviors ("Alice is always chatting while

Bob is all about work”), which makes the simulation feel richer. From an implementation standpoint, we’ll need to ensure the policy/agents can incorporate these differences (e.g. through observation or reward function adjustments) and reflect them in the dashboard (maybe an icon or note on each agent’s card indicating their trait). With personalities in place, the town’s story will naturally become more varied and interesting.

- **Personal Agent Inventories & Items:** Expand the world model to include **objects and inventories** that agents can carry or use, adding a layer of depth to agent interactions. Currently, agents have a wallet for money and counters for things they’ve done (meals eaten, etc.), but no tangible personal items. We will allow agents to pick up and hold items – for example, food portions, tools, or other resources – in a simple inventory system. This enables new gameplay mechanics: an agent might cook a meal and *store it* for later, or pick up a tool (e.g. a book, a toy) that could satisfy a need like entertainment or education if we introduce those. Inventories create opportunities for **trading or sharing** behaviors as well: one agent could give an item to another (think of a neighbor lending sugar, or a friend sharing a meal), which can strengthen social ties and produce heart-warming emergent stories. We’ll start with one or two item types tied to existing needs – e.g. a “meal item” that an agent can carry and eat later (instead of eating immediately at the source). The world objects (like the kitchen or store) would produce these items (cooking yields a meal item in inventory instead of instantly consuming it). We’ll track inventory in the agent’s state and update observations so that agents know what they are carrying. The UI will be updated to show if an agent has something in their inventory (for instance, an icon next to their name if they have a meal or tool). While this adds some complexity, it aligns with the “Sims-like” feel – agents having personal belongings. It can lead to emergent outcomes such as agents planning for the future (stockpiling food if prices are rising) or social outcomes (an agent with extra food might become popular). Overall, implementing inventories and simple item economics will make the simulation more **interesting technologically** and narratively, as agents interact not just through needs and money, but also through tangible goods.

- **Richer Social Interactions & Relationships:** Build on the social framework to make agent interactions more engaging and observable. The groundwork for relationships is in place (trust and rivalry metrics that update on events like shared meals [35]). To enrich this, we’ll introduce more varied social affordances and consequences. For example, implement a **“conversation” or “chat” affordance** that agents can use during free time, which increases familiarity or trust between them. We can leverage the existing hooks (like a `force_chat` command exists for the palette) to allow organic chats to occur when agents bump into each other or have downtime. Include a chance of **negative interactions** too – e.g. if two rivals end up in a queue together, a conflict event could trigger (reducing trust or raising rivalry). We’ll also add **social “stories”** or mini-narratives: small scripted vignettes that play out under certain conditions (for instance, if an agent’s trust with another goes above 0.8, trigger an event “they become best friends and start spending time together”, affecting their behavior; or if rivalry goes too high, trigger a confrontational event or an avoidance behavior). The output of these should be visible: e.g. a narration line “Alice and Bob have become friends!” or an entry in the social event feed panel [36]. Technically, this means extending the event emitter and narrative logging system to handle these new social events, and ensuring the **observer dashboard surfaces them** (perhaps highlighting the affected agents in the UI or listing recent social events). By layering in these social interactions, we aim for **emergent group behaviors** – cliques forming, feuds simmering – much like stories that come out of The Sims. It’s important to throttle and tune these so they feel natural and don’t overwhelm the simulation with constant events [37]. With careful design, these enhancements will let the audience witness unscripted social dynamics (friendships, rivalries, alliances) that make the town simulation truly compelling.

- **Extended Visualization & Feedback:** To match the increasing complexity of the simulation, invest in visualization enhancements. This involves both improving the existing console dashboard and planning for future graphical interfaces:

- In the console UI, incorporate the new features into the display. For instance, if agents have inventories or distinct personalities, add that info to their agent cards or tooltips. We might include a small **inventory list per agent** (or at least an icon if they have something notable, like a meal icon if carrying food). For personalities, perhaps color-code name tags or add an initial (e.g. “[S]” for social, “[C]” for competitive) in the agent table to remind viewers of their traits. If we implement new needs or stats (like an entertainment need or a stress level), add those as bars or indicators in the UI as well. Essentially, keep the UI **in sync with simulation complexity** – everything important an agent is doing or feeling should be observable in some form [38].
- Enhance the map/visual representation of the world. Currently, the dashboard shows an egocentric local map for one agent [39]. We can improve this by allowing the viewer to switch the focused agent or to display a global map if feasible. A simple top-down grid view of the townlet with icons for agents and objects would greatly help viewers track movements and gatherings. Since a fully graphical 3D view is out of scope (per non-goals [40]), we can aim for an ASCII or minimal 2D representation. If time permits, integrate a lightweight web visualizer that subscribes to telemetry – for example, a web page that draws the grid and updates agent positions in real-time. This could be an extension of the telemetry client (translating JSON positions to a canvas).
- Implement an **event highlighter** in the UI. When notable events happen (perturbation, conflict, friendship, etc.), make them visually salient. The design calls for a perturbation banner which we’ll have [41]. We can also flash or highlight an agent’s card when they experience something significant (like a red outline if an agent is in conflict, or a green highlight if they fulfill all needs). Providing visual feedback loops like this ensures that as emergent behaviors occur, the audience notices them.
- Consider audio or simple notifications as well (pending the earlier audio toggle decision). Even without full audio simulation, a few sound cues (beeps or text-to-speech narrations) could be fun for a live demo – e.g. a bell sound when a new agent arrives, or a buzzer if an agent’s needs hit zero. This must be toggleable for accessibility [21], but it adds another layer of immersion.

Overall, this work package is about making sure **what the agents do is visible and understandable** to observers. As we layer on complexity (events, items, personalities), we don’t want to overwhelm the viewer; instead we present it in an intuitive dashboard. The end result should feel like a **mini “Sims” interface** – with panels for needs, relationships, and events – allowing the tech demo audience to appreciate the nuanced behaviors and stories unfolding in Townlet.

- **Polish, Testing, and Tuning:** Finally, allocate time for polishing these enhancements and tuning the simulation. As new features (perturbations, inventory, etc.) are added, we’ll extend the test suites (unit tests for new affordances, integration tests for event sequences) to maintain stability. We’ll run long simulations to see if any unintended behaviors emerge (e.g. do agents starve if too many events happen? Does the social network stabilize?). Use the telemetry and KPIs defined in the charter to monitor success: for instance, ensure that even with perturbations, at least ~70% of agents keep their needs satisfied over long runs [12], that job attendance stays high despite events, and that most agents form relationships by the end of the demo scenario. Any tweaks to rewards or agent logic to achieve these can be done in this phase. We’ll also address lower-priority tech debt or nice-to-haves if time allows (for example, cleaning up affordance outcome logging [42] [43] or refining the world state architecture for maintainability). The polish step includes updating all documentation – the Ops handbook, README, and in-line docs – so that internal stakeholders and future contributors understand the new features and how to use them during the demo. By the end of this, Townlet should be **demo-ready and robust**: a small town simulation where deep RL agents pursue daily routines, respond to surprises, and create emergent little stories, all visible through a rich interface. This will fulfill our goal of a captivating tech demo that not only proves out the technology but genuinely delights its viewers [9] .

---

[1] [2] [3] [4] [5] [6] [7] [10] [11] [13] [14] [15] [42] [43] WORK_PACKAGES.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/audit/WORK_PACKAGES.md

[8] [9] [12] [28] [40] PRODUCT_CHARTER.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/PRODUCT_CHARTER.md

[16] [17] [29] [30] WORK_PACKAGES.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/audit.old/WORK_PACKAGES.md

[18] [19] [36] [37] [38] [39] [41] DASHBOARD_UI_ENHANCEMENT.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/design/DASHBOARD_UI_ENHANCEMENT.md

[20] [21] [22] [27] [31] [34] MASTER_PLAN_PROGRESS.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/MASTER_PLAN_PROGRESS.md

[23] [24] [25] [26] DEMO_PLAYBOOK.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/ops/DEMO_PLAYBOOK.md

[32] [35] default.py
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/src/townlet/world/hooks/default.py

[33] MILESTONE_ROADMAP.md
https://github.com/tachyon-beep/townlet/blob/ce19ff3ed21fa8e36cb74c402d72b52eeb93a8c4/docs/program_management/MILESTONE_ROADMAP.md