ChatGPT

Executive Overview

VMT (“Visualizing Microeconomic Theory”) is a **spatial agent-based simulation** framework for studying how market phenomena emerge from micro-level foraging and exchange interactions [1] [2]. Each agent occupies a cell on a grid, holds inventories of goods (A, B, and optionally money M), and follows a utility-maximizing behavior. Agents search locally for trade or forage opportunities, form bilateral matches, and negotiate exchanges. Prices and allocations **emerge endogenously** from explicit institutional rules (search, matching, bargaining) rather than being assumed in advance [3] [4].

VMT’s **core architecture** enforces strict **determinism** and modularity. A canonical “Protocol→Effect→State” pattern ensures that *protocol modules* (search, matching, bargaining) never mutate state directly, but instead return declarative *Effects* which the simulation applies in a fixed order [5] [6]. Each tick (time step) proceeds through a **7-phase cycle** in a fixed order [7] [8]: *Perception*, *Decision (pairing)*, *Movement*, *Trade*, *Forage*, *Resource Regeneration*, and *Housekeeping*. For example, in **Phase 2 (Decision)** agents build preference lists (using distance-discounted surplus) and pair via a three-pass algorithm [9]; in **Phase 4 (Trade)** committed pairs search for a feasible trade block via a compensating-block price search [10]. Determinism is guaranteed by global rules: “All loops over agents are sorted by `agent.id`,” “trade pairs sorted by (min_id, max_id),” and **fixed tie-break rules** (e.g. move along x before y on ties) [11] [12]. A strong **type system** (see Type Overview [13]) defines data invariants for agents, grid cells, and scenario parameters, ensuring validity (e.g. Stone-Geary subsistence constraints) before simulation.

The repository is organized into distinct components [14]:

```
vmt-dev/
├── docs/                  # Project and system documentation
├── scenarios/             # User scenario YAML files (built-ins and demos)
├── src/
│   ├── vmt_engine/        # Core simulation engine (state, agents, systems,
protocols)
│   ├── vmt_launcher/      # GUI launcher (PyQt) for running scenarios
│   ├── vmt_log_viewer/    # GUI telemetry viewer
│   ├── vmt_pygame/        # Pygame visualization renderer
│   ├── telemetry/         # Telemetry logging (SQLite) and config
│   └── scenarios/         # Scenario loader and schema
├── tests/                 # Test suite (protocols, utilities, simulation)
├── main.py                # CLI entry point (runs Pygame visualization)
└── launcher.py            # Alternative GUI entry point
```

Major modules include: the **Core engine** (`vmt_engine/`) with state (`core/`), utility functions, and ordered “systems” implementing each tick phase; a **Protocol** subsystem (`vmt_engine/protocols/`) providing pluggable Search/Matching/Bargaining classes that output *Effect* objects; **Scenarios** (YAML files plus a schema/loader) for configuring agents and parameters; and **Telemetry** (`telemetry/`) that logs simulation data (agent snapshots, trades, pairings, etc.) to a SQLite database [15]. The **Visualization** components (`vmt_pygame/` and GUI) display real-time agent movement, with overlays for pairings, trade arrows, and other state.

This guide leads the learner through modules of increasing depth – from a quick orientation to running simulations, into the deterministic engine and protocol architecture, through scenario design and telemetry, and finally to advanced extensions. Each module provides reading guides, exercises, and hands-on labs, culminating in a capstone project. References and indices at the end map economic concepts (utility, surplus, etc.) to code classes, and list tests and common pitfalls.

Module A: Orientation and Quickstart

Learning Objectives: Understand VMT's purpose and capabilities; run a simple simulation; inspect basic outputs.
Acceptance Criteria: Describe VMT's goals (micro-foundations of markets) and architecture; successfully launch at least one built-in scenario via CLI and GUI; observe deterministic output given a fixed seed.

Prerequisites: Python 3, basic command-line usage.

Key Readings (in order):
- **README.md** (project description and quick start) [2] [16].
- **docs/1_project_overview.md** (overview and quickstart examples) [1] [16].
- **docs/structures/minimal_working_example.yaml** (skeleton scenario).

Guided Prompts & Questions:
- *What phenomenon does VMT simulate?* (Answer: market formation from bilateral search, matching, and bargaining [3].)
- *How do I run a scenario from CLI? What options exist?* (See [main.py], e.g. `python main.py scenarios/three_agent_barter.yaml 42` [16] [17].)
- *What GUI features are available?* (Launcher with scenario browser, visualization controls in [main.py] output [18] [19].)
- *How is reproducibility ensured?* (By using a fixed seed – same seed = identical run [20] [12].)

Exercises:
1. **CLI Run:** Launch a simple scenario in terminal.

```
# Example: three-agent barter scenario with seed 0
python main.py scenarios/three_agent_barter.yaml --seed 0
```

- *Expected Outcome:* A Pygame window appears showing agents. In console, see "Loading scenario..." and control hints [21]. After pressing `Q` to quit, confirm "Final tick: X" and "Logs saved to ./logs/" messages.
- *Validation:* Ensure agents moved and traded plausibly (e.g. agent inventories changed in logs).

1. **GUI Launcher:** Start the GUI application for scenario management.

```
python launcher.py
```

2. *Expected Outcome:* A PyQt window listing scenarios. Try selecting "three_agent_barter.yaml", set seed and run.

3. *Validation:* Simulation runs with visualization. (If a GUI cannot be run in your environment, validate by successful process exit and log creation.)

4. **Seed Invariance:** Run the same scenario twice with the same seed and verify identical results.

   ```
   python main.py scenarios/three_agent_barter.yaml --seed 42
   python main.py scenarios/three_agent_barter.yaml --seed 42
   ```

   Compare the generated telemetry databases (`logs/telemetry.db`) or output logs. *Expected:* No differences.

Hands-On Lab A: *"Hello, VMT!"*
1. Activate the Python virtual environment:

```
cd vmt-dev
python3 -m venv venv && source venv/bin/activate
pip install -r requirements.txt
```

2. Run the minimal example:

```
cp docs/structures/minimal_working_example.yaml scenarios/my_test.yaml
python main.py scenarios/my_test.yaml --seed 123
```

3. In a new terminal, query the telemetry for final agent utilities:

```
sqlite3 logs/telemetry.db "SELECT tick, COUNT(*) FROM trades GROUP BY tick;"
```

Expected: A table showing how many trades occurred per tick (or 0 if none).
4. (Optional) Open the GUI and toggle target arrows (key **T**). Confirm UI responds without error.

Common Pitfalls:
- Not activating the virtual environment or missing dependencies (PyQt6/Pygame).
- Incorrect scenario path.
- Forgetting `--seed` leads to non-determinism across runs.
- Large windows on high-resolution screens may hide UI elements (use arrow keys to scroll or resize).

Evidence of Mastery: Able to cite the vision of VMT (markets as emergent) [22], describe the seven-phase tick, and run at least two different scenarios (one barter, one with more agents). Demonstrate via logs that runs are reproducible (same outputs each run).

Module B: Core Engine and Phase System

Learning Objectives: Master the simulation loop and phase mechanics; understand state updates and foraging/trading logic.
Acceptance Criteria: Explain each of the 7 tick phases and their order; locate where agents' positions, inventories, and pairings are updated in the code; run a short simulation and interpret telemetry logs for each phase.

Prerequisites: Module A, basic familiarity with Python code structure.

Key Files/Sections (in order):
- `src/vmt_engine/README.md` (phase descriptions and determinism rules) [8] [12].
- `src/vmt_engine/simulation.py` (class `Simulation`: setup, `step()` loop) [23] [24].
- **Systems** (in `vmt_engine/systems/`): read `PerceptionSystem`, `MovementSystem`, etc., to see how effects are applied.
- **State classes** (`vmt_engine/core/`): inspect `Agent` and `Grid` to understand data structures (e.g. `Agent.inventory`, `Agent.paired_with_id`).

Guided Prompts & Questions:
- *What are the seven phases, and what happens in each?* Use [vmt_engine/README.md] for a concise list [8].
- *How does the code enforce phase order?* (See `Simulation.step()`: it iterates through a list of `self.systems` in fixed order [24] [23].)
- *How are mode-specific phases handled?* (`_should_execute_system` skips Trade or Forage based on current mode [25].)
- *Where are agents moved?* (In `MovementSystem`, which applies `Move` effects to agent positions.)
- *Where does trading occur?* (In `TradeSystem`, which uses the bargaining protocol to emit `Trade` or `Unpair` effects.)

Exercises:
1. **Examine a Tick:** Run a 1-tick simulation programmatically and print state.

```
from vmt_engine.simulation import Simulation
from scenarios.loader import load_scenario
scenario = load_scenario("scenarios/three_agent_barter.yaml")
sim = Simulation(scenario, seed=42)
sim.step()  # execute one tick
for agent in sim.agents:
    print(f"Agent {agent.id}: pos={agent.pos}, inv=({agent.inventory.A},
{agent.inventory.B}), paired={agent.paired_with_id}")
```

Expected: Each agent's position and inventory after Phase 5 (Forage) of tick 0. Agents either remain unpaired or have a partner ID.
2. **Verify Deterministic Ordering:** Modify the loop over agents to process in reverse ID order (in code), rerun the simulation, and check that outputs change. Then restore the order. (Simulate editing `src/vmt_engine/simulation.py` around line 153: sort descending; rerun).
Expected: With reversed order, because of tie-break rules, the outcome differs — proving sorted-by-ID matters.

1. **Resource Harvesting:** Using the same Python console, step until resources regenerate.

```
sim.run(10)
from telemetry.db_loggers import TelemetryManager
con = sqlite3.connect("logs/telemetry.db")
print(con.execute("SELECT tick, COUNT(*) FROM resource_snapshots GROUP BY tick;").fetchall())
```

Expected: Rows showing resources harvested and regrown (non-zero counts on tick 0 and later ticks). Verify that only one agent harvested a given cell per tick.

Hands-On Lab B: *"Peeking under the hood"*
1. **Phase Annotation:** Enable debug logging around each system. In `src/vmt_engine/systems/` (e.g. at the start of `PerceptionSystem.execute`), add `print(f"Phase Perception, tick {sim.tick}")`. Repeat for all systems. Re-run `sim.step()` in a script and observe the order of prints.
2. **Movement Details:** Pick an agent with `move_budget_per_tick=1`. After one tick, check its Manhattan distance moved (should be ≤ 1). Try toggling the move budget in the scenario and rerun to see effect on speed.
3. **Trade Logging:** Enable detailed trade logging by running simulation with `params.log_trades=True` in scenario. After a few ticks, query:

```
sqlite3 logs/telemetry.db "SELECT agent_a, agent_b, tick, pair_type FROM trades;"
```

Expected: A list of trade events between agent pairs, with exchange type ("A_for_B" etc.).

Common Pitfalls:
- Accidentally using unsorted or Python `dict` iteration can break determinism (never rely on arbitrary order) [12] [26].
- Forgetting to flag `inventory_changed = True` after a trade/forage will skip the quote refresh [12] [27].
- Misconfiguring `mode_schedule` may cause expected phases to be skipped.

Evidence of Mastery: Understand and explain deterministic tie-break rules (e.g. MovementSystem breaks diagonal deadlocks by higher ID) [28]. Ability to correlate ticks with logged pairings/trades. Demonstrate that modifying loop order changes output, reinforcing importance of sorted iteration.

Module C: Protocols In Depth

Learning Objectives: Grasp how VMT's institutional *protocols* are designed and implemented; write and register a simple new protocol; ensure it follows determinism rules.
Acceptance Criteria: Explain the "protocol" design pattern (input WorldView, output Effects) [5] [6]; locate where protocols are loaded and invoked; implement a minimal custom protocol following guidelines and integrate it (with tests).

Prerequisites: Module B; comfort with Python classes and interfaces.

Key Files/Sections (in order):
- `docs/onboarding/5_protocol_development_guide.md` (design rules, skeleton example) [5] [29].
- `src/vmt_engine/protocols/base.py` (abstract classes `ProtocolBase`, `SearchProtocol`, etc.) [30] [31].
- `src/vmt_engine/protocols/registry.py` (protocol lookup and metadata – read through but not quoted here).
- **Examples:** `src/vmt_engine/protocols/search/legacy.py` (legacy search) [32] [33], `bargaining/split_difference.py` (baseline bargaining) [34] [35].
- **Tests:** `tests/test_protocol_registry.py` shows expected protocol names and registration [36] [37].

Guided Prompts & Questions:
- *What is a protocol's interface?* (See `ProtocolBase`: must have `name` and `version` properties [26]. Derived classes (e.g. `SearchProtocol`) implement specific methods like `select_target` and `build_preferences` [38] [31].)
- *How are protocol classes registered and chosen?* (`@register_protocol` decorator assigns a name and category; `ScenarioConfig` names (YAML) map to classes via the registry [39].)
- *What must a new protocol return?* (A list of **Effects** only – e.g. `SetTarget`, `ClaimResource`, `Pair`, `Trade`, etc. [40].)
- *Why no direct state mutation?* (This ensures purity/determinism: "same WorldView → same Effects" [26].)
- *What rng may be used?* (Only the provided seeded RNG in `context`; never `random.random()` global.)

Exercises:
1. **Inspect Legacy Search:** Read `LegacySearchProtocol.select_target` [41]. Note how it emits `SetTarget` or `ClaimResource`. List the effect types you see in this method.
2. **Protocol Metadata:** In Python REPL, list all registered search protocols:

```python
from vmt_engine.protocols import list_all_protocols
print(list_all_protocols()["search"])
```

Expected: Contains e.g. `'legacy_distance_discounted', 'random_walk', ...` [36].
3. **Create a Simple Protocol:** Using the skeleton, write a trivial search protocol `RandomSearch` that picks a random visible agent as target. Register it with `@register_protocol(category="search", name="random_choice", ...)`. Add a line `self.rng = context.rng` to use the seeded RNG. (Use the example in the dev guide.)

Hands-On Lab C: *"Protocol Prototype"*
1. **Implement a New Protocol:** In `src/vmt_engine/protocols/search/`, create `random_choice.py` with:

```python
from ..base import SearchProtocol, SetTarget
from ..context import WorldView
from ..registry import register_protocol

@register_protocol(category="search", name="random_choice",
                   description="Choose a random neighbor as target",
                   properties=["deterministic"], complexity="O(V)")
class RandomChoiceSearch(SearchProtocol):
    @property
    def name(self): return "random_choice"
    @property
    def version(self): return "2025.10.01"
    def select_target(self, world: WorldView):
        neighbors = list(world.visible_agents)
        if not neighbors: return []
        # Use seeded RNG to choose
        chosen = neighbors[world.rng.integers(len(neighbors))]
        return [SetTarget(protocol_name=self.name, tick=world.tick,
                          agent_id=world.agent_id, target=chosen.agent_id)]
```

2. **Activate the Protocol:** Modify a scenario (e.g. copy `three_agent_barter.yaml`) to include `search_protocol: "random_choice"`. Run the simulation twice with the same seed.
3. **Verify Determinism:** Confirm that the chosen targets are the same on both runs (despite being "random") by examining logs:

```
sqlite3 logs/telemetry.db
"SELECT agent_id, target_agent_id, tick FROM decisions ORDER BY tick;"
```

Expected: Identical lists of `(agent_id, chosen_partner, tick)` in both runs (protocol is actually random but seeded) [36] [26].
4. **Protocol Rules Check:** Intentionally break determinism (e.g. use `random.choice` without `world.rng`). Re-run and observe that runs now differ. Revert the fix.

Common Pitfalls:
- **State mutation:** Forgetting and mutating world or agent fields inside a protocol (violates design rule – always return Effects only [5]).
- **Unregistered protocol:** Not using `@register_protocol` or mismatched `name` causes "protocol not found" errors or missing default, as tests illustrate [36] [37].
- **Unseeded randomness:** Using Python's global RNG breaks reproducibility. Always use `context.rng`.
- **Iteration order:** If looping over sets/lists of neighbors without sorting or relying on `world.rng`, the output may vary.

Evidence of Mastery: A passing test for the new protocol (e.g. add to `tests/`: verify its `name`, determinism, etc.). Correct YAML integration (scenario runs without errors). Being able to list protocol metadata programmatically:

```
>>> from vmt_engine.protocols import describe_all_protocols
>>> print(describe_all_protocols()["search"]["random_choice"]["description"])
```

should display the description.

Module D: Scenarios and Typing

Learning Objectives: Gain expertise in configuring and validating simulation scenarios via YAML; understand the type schema and invariants.
Acceptance Criteria: Successfully author a custom scenario YAML with both barter and monetary settings; explain mandatory vs optional parameters; identify and fix schema validation errors; describe key type contracts for agents and resources.

Prerequisites: Modules A–C; familiarity with YAML.

Key Files/Sections (in order):
- **docs/structures/README.md** (scenario structure reference, templates) [42] [43] .
- `docs/4_typing_overview.md` (type definitions for IDs, Inventory, Quotes, etc.) [44] [45] .
- `src/scenarios/schema.py` (the Pydantic schema for scenarios).
- **Example scenarios** in `scenarios/` directory (especially `demos/`).

Guided Prompts & Questions:
- *What are the required top-level fields of a scenario?* (Check `schema.py` or templates: e.g. `name`, `N`, `agents`, initial inventories, utilities mix, resource seed) [46] [47] .
- *How do utility mixes work?* (See `utilities.mix` in YAML: a weighted list of types; each agent is sampled accordingly [48] .)
- *What happens if a required field is missing or invalid?* (Loading throws a validation error with a message – try omitting `initial_inventories.B` and observe the error.)
- *What type is* `resource_regen_cooldown`*?* (From types: an integer ≥0.)
- *Stone-Geary constraints:* What invariant must hold for Stone-Geary utilities? (Initial inventories must exceed subsistence levels [49] .)

Exercises:
1. **Load and Lint a Scenario:** Attempt to load a YAML that violates the schema. E.g.:

```
python - <<EOF
from scenarios.loader import load_scenario, SchemaError
try:
    load_scenario("scenarios/invalid_example.yaml")
except SchemaError as e:
    print("Validation failed:", e)
EOF
```

Create `invalid_example.yaml` missing a required field (say, remove `N`). *Expected:* A printed validation error indicating the missing field. Fix the YAML and reload successfully.

1. **Custom Scenario:** Starting from `minimal_working_example.yaml`, create a scenario with 4 agents, some heterogeneous utilities, and a mode schedule (alternate between "trade" and "forage"). Run it for 20 ticks:

   ```
   cp docs/structures/minimal_working_example.yaml scenarios/custom.yaml
   # Edit custom.yaml: set name, N=10, agents=4, define two utility functions
   in mix, add a simple mode_schedule.
   python main.py scenarios/custom.yaml --seed 7
   ```

   Expected: Simulation runs without schema errors. Check `sqlite3 logs/telemetry.db "SELECT mode, tick FROM tick_states"` to see mode transitions.

2. **Parameter Limits:** Try setting an invalid parameter (e.g. `vision_radius: -1`). Load scenario: should raise an error about range. Check in `docs/structures/comprehensive_scenario_template.yaml` how ranges are documented.

Hands-On Lab D: *"Build a Scenario"*
1. **Use the minimal example:**

```
cp docs/structures/minimal_working_example.yaml scenarios/my_scenario.yaml
```

2. **Edit with typical values:** Open `my_scenario.yaml` and set:
- `name: "TestMarket"`
- `N: 12` (grid width)
- `agents: 6`
- `initial_inventories.A/B: 10`
- `utilities.mix:` (one `CES` with `rho=0.5,wA=0.6,wB=0.4` weight 1.0)
- `resource_seed.density: 0.2`, `amount: 5`
- Add a simple mode schedule that trades for 10 ticks then forages for 10 ticks.
3. **Validate and Run:**

```
python main.py scenarios/my_scenario.yaml --seed 99
```

Expected: No errors, simulation runs.
4. **Inspect Telemetry Schema:** Launch the SQLite viewer:

```
python view_logs.py
```

(If GUI not available, use CLI to inspect tables:)

```
sqlite3 logs/telemetry.db "SELECT name FROM sqlite_master WHERE type='table';"
```

Expected: Tables `simulation_runs`, `agent_snapshots`, `trades`, etc.

Common Pitfalls:
- **Indentation/YAML syntax:** Misaligned lists or missing `-` leads to parse errors.
- **Mismatched list lengths:** If specifying lists of inventories, length must equal `agents`. Otherwise schema error (see validation section in docs) [50] .
- **Subsistence violation:** For Stone-Geary utilities, ensure `initial_A > gamma_A` [49] .
- **Missing `schema_version`:** Always include the schema version (e.g. `1`) at top.

Evidence of Mastery: Successfully load and run at least one barter and one monetary scenario. Being able to locate and explain a schema error (e.g. negative vision radius). Demonstrate knowledge of utility config by citing `[12†L49-L58]` for how quotes are defined from reservation bounds.

Module E: Telemetry and Experimentation

Learning Objectives: Utilize VMT's telemetry for analysis; design experiments and compare runs; understand logging internals.
Acceptance Criteria: Explain the database schema (tables for agents, trades, pairings, etc.) [51] ; run simulations with different regimes and extract comparative statistics from the logs; demonstrate deterministic telemetry (same run_id for repeated seeds); use the GUI or SQL to inspect results.

Prerequisites: Modules A–D, basic SQL or familiarity with data analysis.

Key Files/Sections (in order):
- **docs/1_project_overview.md (Telemetry section)** [15] – table of telemetry DB contents.
- `src/telemetry/database.py` (schema definitions for tables, primary keys).
- `src/telemetry/db_loggers.py` (TelemetryManager: how logs are buffered and written).

Guided Prompts & Questions:
- *What tables exist in the telemetry DB?* (From [17]: `simulation_runs`, `agent_snapshots`, `trades`, `pairings`, `preferences`, etc. [51] .)
- *What is logged for each agent snapshot?* (Position, inventory, utility, quotes, target, λ) – see `TelemetryManager.log_agent_snapshots` in `db_loggers.py`.
- *How to access logs programmatically?* (Use `sqlite3` CLI, pandas with SQL, or the built-in `view_logs.py` GUI.)
- *What ensures telemetry matches simulation state?* (Records include `run_id`, tick, and all agents' data; same seed produces identical logs.)

Exercises:
1. **Explore `simulation_runs`:** After running a simulation, inspect the `simulation_runs` table:

```
sqlite3 logs/telemetry.db "SELECT scenario_name, start_time, end_time,
total_ticks FROM simulation_runs;"
```

Verify the scenario name and ticks match your run.
2. **Agent Snapshot Query:** For a given run, find the maximum utility reached by any agent:

```
sqlite3 logs/telemetry.db "
  SELECT MAX(utility_val) FROM agent_snapshots WHERE run_id=1;"
```

Expected: A single float value.
3. **Trade and Surplus:** Compare total surplus in barter vs monetary modes. Run a scenario twice (once with `exchange_regime: barter_only`, once `mixed`) and sum trades' surplus:

```
SELECT SUM((metadata->>'surplus_i') + (metadata->>'surplus_j')) FROM trades;
```

(Use SQLite JSON syntax or extract from `metadata` if stored as JSON.) Observe differences.

Hands-On Lab E: *"Data Mining"*
1. **Run Paired Experiments:** Execute the same scenario with two different seeds or regimes and log to separate DBs (use `--seed` and copy `telemetry.db` after each).

```
python main.py scenarios/three_agent_barter.yaml --seed 1
mv logs/telemetry.db logs/telemetry_seed1.db
python main.py scenarios/three_agent_barter.yaml --seed 2
mv logs/telemetry.db logs/telemetry_seed2.db
```

2. **Compare Key Metrics:** For each DB, use `sqlite3` to compute total trades and mean trade surplus:

```
sqlite3 logs/telemetry_seed1.db "SELECT COUNT(*), AVG((metadata->>'surplus_i'))
FROM trades;"
sqlite3 logs/telemetry_seed2.db "SELECT COUNT(*), AVG((metadata->>'surplus_i'))
FROM trades;"
```

Expected: The two runs (different seeds) yield identical counts and averages (since determinism, if same seed, or differ in a controlled way if different seed).
3. **Interactive Viewer:** Launch the log viewer GUI:

```
python view_logs.py
```

Select a run and step through ticks, inspecting agent states. Confirm that the visualization of trade arrows and agent data matches what you queried via SQL.

Common Pitfalls:
- Forgetting to enable logging (by default, LogConfig.standard() is used). No errors, but empty tables.
- Large DB sizes for long runs – remember to purge or set appropriate snapshot frequencies (config in scenario).
- Mixing logs from different runs – always use fresh or correctly labelled DB.

Evidence of Mastery: Demonstrate extracting non-trivial insights: e.g., plot average utility over time using the snapshots. Show that two runs with same seed have identical timestamped logs, confirming **telemetry identity** across runs [15] [12].

Module F: Advanced Topics and Extensions

Learning Objectives: Explore advanced features and customization: monetary modes, mode scheduling, custom extensions, and performance considerations.
Acceptance Criteria: Describe how the money system is integrated (exchange regimes `barter_only`/ `money_only`/`mixed`), and how λ-updates or KKT mode work; implement an experimental extension (protocol or mode); profile performance; ensure no determinism violations in complex changes.

Prerequisites: Completion of A–E; comfort modifying and testing code.

Key Topics & References:
- **Money System:** See [8†L87-L95] for money-aware API (λ parameter) and how barter vs monetary exchanges are handled in decision and trading.
- **Mode Scheduling:** Scenario parameters (`mode_schedule`) to switch between “trade” and “forage” phases. Code in `Simulation.step` handles this [23] [52].
- **KKT/Lambda Update (Phase 3):** Not fully implemented in main; `HousekeepingSystem` has placeholders. Consider reading planned features sections.
- **Performance:** Spatial indexing is used (KD-tree) [12]. For large N, profile which phase dominates.

Guided Prompts & Questions:
- *How does the mixed exchange regime work?* (Agents compute both barter and money trades; the matching algorithm selects highest-surplus exchange [53] [54].)
- *What happens in the Housekeeping phase?* (Quote refresh, pairing integrity checks, optional λ updates [55] [56].)
- *How to add a new feature without breaking determinism?* (Ensure all loops remain sorted, use RNG only via context, write tests.)

Exercises:
1. **Mode Schedule Test:** Create a scenario with `mode_schedule` that alternates between “forage” and “trade” every tick. Run and observe that when in “forage” mode, no trades happen (check `trades` table).
2. **Monetary Exchange:** Copy a barter scenario, change `exchange_regime: mixed` and add money (`initial_inventories.M`, λ values). Run for 50 ticks. Compare the price convergence or surplus vs. the barter run. Observe money movements in logs (`trades` with type “A_for_M”).

3. **Profiling:** Run a stress test in Python and time it:

```
time python - <<EOF
from vmt_engine.simulation import Simulation
from scenarios.loader import load_scenario
sim = Simulation(load_scenario("scenarios/large_grid.yaml"), seed=1)
sim.run(max_ticks=100)
EOF
```

Identify which phase takes most time (via profiling tools or by inserting timestamps in `Simulation.step`). Experiment with lowering `vision_radius` to speed up search.

Hands-On Lab F: *"Customize and Benchmark"*
1. **Add a New Monetary Mode:** Suppose we want a "mixed_liquidity_gated" regime (in planning). Create a placeholder implementation: in `DecisionSystem` or config, treat it like mixed but with a rule (e.g. require each agent to have at least 1 money to trade). Write tests to verify that with zero money agents do not trade under this mode.
2. **Extension – Tie-Break Protocol (Capstone Preparation):** Read the Capstone section. Begin designing or partially coding a novel protocol (e.g. a deterministic trade tie-breaker).
3. **Performance Logging:** In `Simulation.run()`, instrument timing around each system execution, logging to console. Run a few ticks and verify that e.g. *Decision* or *Trade* is the bottleneck.

Common Pitfalls:
- **Side-effects in extensions:** Accidentally using floating-point comparisons unsafely can introduce non-determinism (e.g. `random` in C). Stick to seeded randomness.
- **Breaking type invariants:** When adding modes or goods, ensure scenario schema and code invariants (e.g. currency units) are updated, or validation will catch it.
- **Forgetting to clear claims:** Resource claims must be released each tick; custom code must respect that.

Evidence of Mastery: Demonstration of at least one custom extension working correctly and deterministically. For instance, if adding "mixed_liquidity_gated," show that trades occur only when both parties have money above a threshold, and that identical seeds yield identical outcomes. Report on performance (e.g. "DecisionSystem took 70% of runtime on average, as expected for an O(N^2) pairing step.").

Capstone Project: New Deterministic Trade Protocol

Objective: Implement a **"Halved-Price Negotiation"** protocol: when two agents are paired, they always trade one unit at a price equal to the midpoint of their reservation bounds. This enforces a fair split of surplus and is fully deterministic.

Description: Create `HalvedPriceBargain` in `src/vmt_engine/protocols/bargaining/halfbid.py`, registered as `name="halfbid"`. The protocol should:

- For a given agent pair, compute each agent's reservation (seller's ask and buyer's bid) from their `WorldView`.
- Set `price = (ask + bid)/2`.
- Attempt to trade ΔA = 1 unit (one good A for B or money as per regime) at this price.
- If both get positive utility gain (use the existing utility API), emit a `Trade` effect. Otherwise emit `Unpair` with reason `"no_gain"`.

Implementation Sketch: (excerpt)

```
@register_protocol(category="bargaining", name="halfbid", description="Midpoint
pricing for 1 unit", ...)
class Halfbid(BargainingProtocol):
    @property
    def name(self): return "halfbid"
    @property
    def version(self): return "2025.10.30"
    def negotiate(self, pair, world: WorldView):
        i, j = pair
        # Build pseudo-agents (or use world) to compute ask/bid for each
        ask = world.quotes.get("ask_A_in_B", 0.0)
        bid = world.quotes.get("bid_A_in_B", 0.0)
        price = 0.5 * (ask + bid)
        dA = 1
        # Compute resulting inventories
        # (Simplified: treat i as buyer, j as seller for illustration)
        # Check both ΔU > 0
        # If good: return [Trade(...)] else [Unpair(...)]
```

(Full code would adapt to A↔B or A↔M context.)

Determinism and Correctness Test:
Write a unit test (e.g. in `tests/test_halfbid.py`) that:

```
from vmt_engine.protocols import ProtocolRegistry
from vmt_engine.simulation import Simulation
from scenarios.loader import load_scenario

# Use a simple two-agent scenario where a known trade should occur
scenario = load_scenario("scenarios/halfbid_demo.yaml")
sim = Simulation(scenario, seed=1, log_config=LogConfig.debug())
sim.run(max_ticks=1)
# After one tick, check inventories obey the half-price trade
agent0, agent1 = sim.agents
assert agent0.inventory.A == original_A0 - 1
assert agent1.inventory.A == original_A1 + 1
# Further checks on B or M, and that trade.price was (ask+bid)/2.
```

This proves the protocol yields expected exchange. Run twice to confirm same result (determinism).

Scenario Demonstration (YAML): `scenarios/halfbid_demo.yaml`

```yaml
name: half_price_example
N: 5
agents: 2
initial_inventories:
  A: [5, 1]
  B: [0, 5]
utilities:
  mix:
    - type: "UStoneGeary"
      weight: 1.0
      params: {alpha_A: 1.0, alpha_B: 1.0, gamma_A: 0.0, gamma_B: 0.0}
params:
  trade_cooldown_ticks: 0
bargaining_protocol: "halfbid"
# (Include any required money fields if using A<->M trades.)
```

In this scenario, agent 0 has mostly A, agent 1 mostly B, so a trade of 1 A for some B at the mid-price is always beneficial.

Performance Benchmark:
Instrument `Simulation.step()` with timestamps (e.g. Python's `time.perf_counter()`) to log durations of each system. Run a large-scale scenario (e.g. 100 agents) with the new protocol:

```
python - <<EOF
import time
from vmt_engine.simulation import Simulation
from scenarios.loader import load_scenario
sim = Simulation(load_scenario("scenarios/large_trading.yaml"), seed=42)
start = time.perf_counter()
sim.step()
mid = time.perf_counter()
sim.step()
end = time.perf_counter()
print("Step1:", mid-start, "Step2:", end-mid)
EOF
```

Expected: Times for the bargaining phase under `halfbid` should be modest (only one price calc per pair) compared to legacy. Logging should show trade events with price exactly at midpoint.

Economic & Design Note: This protocol embodies a **fair division** principle (akin to an approximate Nash bargaining solution) by splitting the difference in reservation prices [57]. It uses VMT's declarative Effects (`Trade` or `Unpair`), preserving the modular design. Midpoint pricing is a simple economic idea (agents meet halfway between their willingness-to-pay and willingness-to-sell), implemented here as a deterministic algorithm that can be compared against the legacy "first acceptable" method.

Reference Sections

Glossary (Economics ↔ Code)

- **Utility function**: VMT supports CES, Linear, Quadratic, Translog, Stone-Geary (classes in `vmt_engine/econ/utility.py`) [58] [59]. E.g. `UStoneGeary` maps to file `src/vmt_engine/econ/utility.py`.
- **Marginal Rate of Substitution (MRS)**: Used in reservation bounds (`u.mrs_A_in_B`, code in `src/vmt_engine/econ/utility.py`).
- **Reservation Price**: Price below which a seller won't sell (p_min) or above which a buyer won't buy (p_max) [60]. Represented in code by `agent.quotes["p_min"]`, `agent.quotes["p_max"]` (State model in `core/state.py`).
- **Pairing (Match)**: A bilateral commitment between two agents (`Pair` / `Unpair` effects in `protocols/base.py` [61]). Implemented via `DecisionSystem` using `SearchProtocol` + `MatchingProtocol`.
- **Surplus**: Sum of utility gains from a trade. Computed via `find_compensating_block_generic` in `systems/matching.py` (Phase 3) [62] [63].
- **Foraging**: Harvesting resources yields ΔUtility = U(A+ΔA,B)−U(A,B). Handled in `ForageSystem` and logged as `Harvest` effects [64].
- **Lambda (λ)**: Marginal utility of money (used in quasilinear preferences). Stored per agent (`agent.lambda_money`) and updated optionally (KKT mode) [65].

Repository Map

- `src/vmt_engine/core/`: Core data models (Agent, Inventory, Grid). See `Agent` in `core/agent.py`, which holds `id, pos, inventory, quotes, target, pairing state` [66] [67].
- `src/vmt_engine/systems/`: Phase logic modules (Perception, Decision, Movement, Trading, Foraging, Regeneration, Housekeeping). Each has an `execute(sim: Simulation)` method. For example, `MovementSystem` reads each agent's `target_pos` and issues `Move` effects.
- `src/vmt_engine/protocols/`: Abstract protocol interfaces and implementations. Subdirectories: `search/`, `matching/`, `bargaining/`. Legacy algorithms (distance-discounted search, three-pass matching, compensating-block bargaining) live here (e.g. `legacy.py`). New protocols (e.g. `random_walk`, `split_difference`) are registered here.
- `src/vmt_engine/simulation.py`: Orchestrates setup and tick loop. Registers systems and protocols, initializes agents, spatial index, and TelemetryManager.

- `src/scenarios/`: Scenario loader and schema (`schema.py`) using Pydantic. Responsible for parsing and validating YAML into `ScenarioConfig`.
- `src/telemetry/`: Telemetry logging engine. `TelemetryManager` (in `db_loggers.py`) batches data to `telemetry.db` (schema in `database.py`).
- `vmt_pygame/` **and** `vmt_launcher/`: Visualization code (not core to simulation logic).
- `tests/`: Unit tests for each subsystem. Key tests: `test_protocol_registry.py` (protocol registration) [36], plus tests for math utilities, scenario validation, and integration tests. (E.g. tests confirming barter trades, foraging, etc.)

Test Index

Major test categories include:
- **Protocol Tests**: `test_protocol_registry.py` verifies registry and metadata [36] [37]. Ensure all expected protocol names exist.
- **Utility/Math Tests**: Validate utility functions (e.g. CES, translog) and edge cases (e.g. zero-inventory handling).
- **Simulation Invariants**: Tests that repeated runs with same seed produce bit-identical telemetry snapshots.
- **Scenario Validation**: Tests that invalid YAML configs raise errors (schema enforcement).
- **Integration Tests**: End-to-end scenarios (demos) where the final agent distribution or trade count is known.
- **Telemetry Tests**: E.g. verify that TelemetryManager correctly logs to the database schema (if present).

Common Violations & Detection

- **Nondeterministic Looping**: Accidentally iterating an unordered collection. *Detection:* Run the same scenario twice; if outputs differ without randomness, check iteration order. Code review: ensure `sorted()` is used for agents and pairs [11].
- **State Mutation in Protocols**: Modifying `world` or agent state in a protocol (contravenes design). *Detection:* Protocol code reviews or failing immutability checks. Use linter or review `@register_protocol` implementations to ensure only Effects are returned [5] [6].
- **Integer vs Float Errors**: Prices and utilities are floats, inventories are int. Mistakes like fractional inventory. *Detection:* Monitoring `inventory` fields (schema requires int) or failing conservation tests. The engine enforces integer rounding (round-half-up) in `find_compensating_block_generic` [63].
- **Disallowed Interactions**: E.g. two agents forging on same resource or exceeding `vision_radius`. *Detection:* Check logs for simultaneous `Harvest` effects on one cell – should never happen when `enforce_single_harvester=True`.
- **Telemetry Drift**: If run states diverge, identify where logs differ. Use snapshot comparison (e.g. diff the `telemetry.db` from two runs).

Mastery Checklist

- [ ] **7-Phase Order**: Confirm the engine always follows Perception→Decision→Movement→Trade→Forage→Regeneration→Housekeeping in each tick (check code and logs) [7] [23].

- [ ] **Deterministic Iteration**: All loops over agents/pairs use sorted lists (no `dict` iteration). Insert additional assertions in code or review loops to ensure sorting by `(min_id, max_id)` for pairs [11] [12].
- [ ] **Effects-Only Updates**: Verify that every custom protocol returns only `Effect` objects and does not directly change `sim.agents` or `sim.grid`. (Protocols should use `WorldView` and return a list of e.g. `Move`, `Trade`, `Pair`, etc.) [5] [6].
- [ ] **Typed State Invariants**: Check that initial scenario parameters satisfy types (e.g. `N, vision_radius` are int≥0, inventories non-negative) [68] [69]. Stone-Geary subsistence constraints hold.
- [ ] **Telemetry Consistency**: Given same seed and scenario, telemetry entries (agent positions, inventories, trades) must match run-to-run [70] [15]. Use a database diff or checksums.
- [ ] **Logging Checks**: All agent inventory changes set `inventory_changed=True` so quotes refresh only in housekeeping [12]. Ensure no quotes are updated mid-tick.
- [ ] **Trade and Pairing Rules**: Confirm successful trades keep `paired_with_id`, failed trades issue `Unpair` with appropriate reason (no_fefasible_trade or trade_failed). Check cooldowns apply (if two agents repeatedly fail, they should cooldown) [71] [72].
- [ ] **Common Pitfall Awareness**: Be able to spot misuses like unseeded randomness, missing `@register_protocol`, or missing `lambda_money` for money scenarios, and know how to detect them (e.g. schema errors, nondeterminism tests).

Completion of this checklist, along with successful lab exercises and capstone results, indicates deep implementation fluency in the VMT repository.

---

1 3 15 16 18 20 22 51 70 1_project_overview.md
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/docs/1_project_overview.md

2 README.md
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/README.md

4 7 9 10 11 14 48 49 53 54 55 58 60 62 63 65 2_technical_manual.md
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/docs/2_technical_manual.md

5 29 5_protocol_development_guide.md
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/docs/onboarding/5_protocol_development_guide.md

6 26 27 30 31 38 40 61 64 71 72 base.py
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/src/vmt_engine/protocols/base.py

8 12 28 README.md
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/src/vmt_engine/README.md

13 44 45 47 59 66 67 68 69 4_typing_overview.md
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/docs/4_typing_overview.md

17 19 21 main.py
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/main.py

23 24 25 52 56 simulation.py
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/src/vmt_engine/simulation.py

32 33 41 legacy.py
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/src/vmt_engine/protocols/search/legacy.py

34 35 57 split_difference.py
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/src/vmt_engine/protocols/bargaining/split_difference.py

36 37 test_protocol_registry.py
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/tests/test_protocol_registry.py

39 protocol_factory.py
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/src/scenarios/protocol_factory.py

42 43 46 50 README.md
https://github.com/cmfunderburk/vmt-dev/blob/6ed9b73283ba3c96c823d866112dc080bac1c00e/docs/structures/README.md