Testrist
Testing Documentation and Known Issues

Robert Rice

December 5, 2025

1. Testing Documentation

1.1 Testing Strategy

The goal of testing for **Testrist** is to ensure that the game meets its functional requirements and that the leaderboard backend behaves correctly and safely.

We used a mix of:

- **Black-box testing**
  These tests focus on the game from the player's point of view. They cover piece spawns, player controls, collision detection, line clearing, scoring, level progression, game over, and the start/pause/reset controls. They are implemented as end-to-end tests using **Playwright**, which automates a real browser and interacts with the UI like a user would.
- **White-box testing**
  These tests focus on the internal behavior of the Python FastAPI leaderboard backend. They call the API endpoints directly, check HTTP responses, and verify the database state in the SQLite database.

For each test, we record:

- Inputs (key presses, button clicks, API payloads, etc.),
- Expected outputs (visual changes, API responses, database effects),
- Requirement IDs (for black-box tests),
- Source code references (for white-box tests),
- Pass/Fail status when the test is actually run.

This structure makes it easy to re-run the same tests after any code change (regression testing) and to show how our tests are connected to both the requirements and the source code.

1.2 Requirements and Traceability

To keep traceability clear, we assigned simple requirement IDs.

1.2.1 Game Requirements (Frontend)

- **G1.0 Tetromino Spawns**
  - G1.1: A tetromino spawns when a new game starts.
  - G1.2: A new tetromino spawns after each piece locks.
  - G1.3: The next tetromino is chosen pseudo-randomly.
- **G2.0 Player Controls**
  - G2.1: Player can move the piece left and right within the board.
  - G2.2: Player can soft-drop and hard-drop the piece.
  - G2.3: Player can rotate the piece, and rotation respects walls and the existing stack.
- **G3.0 Collision Detection**
  - G3.1: Falling pieces stop when they hit the stack or floor.
  - G3.2: Pieces cannot move out of bounds.

  - G3.3: Rotations cannot overlap existing blocks.
- **G4.0 Line-Clearing Logic**
  - G4.1: A fully filled horizontal line is cleared.
  - G4.2: Multiple lines can be cleared at once (double, triple, "Tetris").
  - G4.3: Blocks above cleared lines fall down correctly.
- **G5.0 Scoring System / Leaderboards**
  - G5.1: Score increases according to the number of lines cleared.
  - G5.2: Score is displayed while the game is running.
  - G5.3: Final score is shown on the game over screen.
  - G5.4: Player can submit a score to the leaderboard.
  - G5.5: Leaderboard lists top scores in the correct order.
- **G6.0 Level Progression**
  - G6.1: Level increases after a defined number of cleared lines.
  - G6.2: Piece fall speed increases with each level.
- **G7.0 Game Over**
  - G7.1: Game ends when a new piece cannot spawn due to filled cells.
  - G7.2: Game over screen is shown and normal movement input is disabled.
- **G8.0 Start / Pause / Reset**
  - G8.1: Start button begins a new game.
  - G8.2: Pause button freezes piece movement and timers.
  - G8.3: Resume continues the game from a paused state.
  - G8.4: Reset clears board, score, and level and starts fresh.

1.2.2 Line Clearing and Gravity

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–006	G4.1, G3.1	Confirm single line clear works.	Fill exactly one full row with pieces and let the last piece lock.	Full row disappears; rows above move down by one; cleared row is empty.	
BB–007	G4.2, G4.3	Confirm multi-line clear (for example, a "Tetris").	Set up board so that one piece clears 2–4 full rows at once.	All completed lines are removed together; score and line count reflect a multi-line clear; blocks above fall correctly.	
BB–008	G4.3	Ensure gravity behaves after partial clears.	Create a pattern where clearing a lower line should make a group of blocks drop straight down.	After the clear, blocks fall straight down with no diagonal shifts or floating gaps.	

Each black-box test case lists the Gx.y requirement(s) it covers. Each white-box test case lists the Lx.y requirement(s) and the related source code element.

1.3 Test Environment and Tools

- **Frontend:** A React-based Testrist game running in a web browser.
- **Backend:** A Python FastAPI application with an SQLite database.
- **Automation tools:**
  - Playwright for end-to-end UI tests (black-box).
  - A Python test client (for example, pytest + FastAPI test client) for API and database verification (white-box).
- **Browsers:** Chromium (primary test target); optional tests in Firefox or WebKit as needed.

1.4 Black-Box Test Cases (Playwright)

The following test cases are implemented as Playwright scripts that open the game, click buttons, and send key presses just like a real player.

1.4.1 Core Gameplay and Controls

ID	Req.	Rationale	Inputs (Actions)	Expected Output	P/F
BB–001	G1.1, G8.1	Verify that the game starts with a piece on the board.	Open game URL; click *Start*.	A tetromino appears at the top spawn row; game state is "running".	
BB–002	G1.2, G3.1	Make sure a new piece spawns only after the old one locks.	Start new game; soft-drop or wait until the current piece reaches the bottom and locks.	Current piece locks into the stack; a new tetromino spawns at the top.	
BB–003	G2.1, G3.2	Test left/right movement and wall boundaries.	Start; repeatedly press *Left* until at left wall, then press *Left* again; repeat on right side with *Right*.	Piece moves until it touches board edges; extra key presses do not move it out of bounds.	
BB–004	G2.2, G3.1	Verify soft-drop and hard-drop behavior.	Start; hold *Down* to soft-drop; in a new game press hard-drop key once.	Soft-drop moves piece faster but stepwise; hard-drop moves piece to lowest valid position and locks immediately.	
BB–005	G2.3, G3.1, G3.3	Check rotation behavior near walls and stack.	Start; move piece next to left wall and press *Rotate*; drop on stack and attempt rotations on top of blocks.	Rotations that would overlap walls or stack are blocked or adjusted; no overlapping blocks appear.	

1.4.2 Line Clearing and Gravity

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–006	G4.1, G3.1	Confirm single line clear works.	Fill exactly one full row with pieces and let the last piece lock.	Full row disappears; rows above move down by one; cleared row is empty.	
BB–007	G4.2, G4.3	Confirm multi-line clear (for example, a "Tetris").	Set up board so that one piece clears 2–4 full rows at once.	All completed lines are removed together; score and line count reflect a multi-line clear; blocks above fall correctly.	
BB–008	G4.3	Ensure gravity behaves after partial clears.	Create a pattern where clearing a lower line should make a group of blocks drop straight down.	After the clear, blocks fall straight down with no diagonal shifts or floating gaps.	

1.4.3 Scoring and Level Progression

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–009	G5.1, G5.2	Verify scoring for single and multi-line clears.	In one game, clear a single line; in a fresh game, clear two lines at once; compare scores.	Score increases after each clear; multi-line clear yields more points than a single line; display updates immediately.	
BB–010	G5.3, G7.1, G7.2	Show final score on game over screen.	Play until the grid is filled and game over is triggered.	Game stops; game over screen shows final score matching last in-game score; movement controls are disabled.	
BB–011	G6.1, G6.2	Confirm level progression and increased fall speed.	Start; clear enough lines to trigger a level-up (for example, ten lines).	Level indicator increases; piece fall interval is shorter after the level-up.	

1.4.4 Game State Controls: Start, Pause, Reset

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–012	G8.1, G8.2, G8.3	Test pause and resume behavior.	Start; let piece fall; click *Pause*; wait a few seconds; click *Resume*.	While paused, piece does not move and controls do nothing; after resume, falling continues from the same position and controls work again.	
BB–013	G8.4, G5.2, G6.1	Test full reset behavior.	Start; play until score, lines, and level are all non-zero; click *Reset*.	Board is cleared; score, lines, and level are set back to starting values; game is ready for a new run.	

1.4.5 Leaderboard via UI (End-to-End)

ID	Req.	Rationale	Inputs	Expected Output	P/F
BB–014	G5.4, G5.5, L1.1, L2.1	Ensure that players can submit scores and see them sorted correctly.	Finish a game; enter a name; submit score; open leaderboard screen.	Leaderboard shows the new entry with correct name, level, lines, and score, in correct sorted order.	

1.5 White-Box Test Cases (Backend API)

To avoid crowding, the white-box tests are summarized in a four-column table: ID, requirements, a short description (including source, inputs, and rationale), and expected output.

1.6 Regression Testing

Any time we change core game logic (spawning, collisions, scoring, levels, game over) or leaderboard behavior, we re-run the tests:

- All UI tests BB–001 through BB–014.
- All API tests WB–001 through WB–006.

For each test run, we keep a simple log that includes:

- Date of the run,
- Version tag of the build (for example, `v1.3.0`),
- List of test cases executed,
- Any failures and a pointer to the related bug or issue ID.

This helps us to repeat the same test sets later, after bug fixes or refactoring.

ID	Req.	Test Description	Expected Output
WB–001	L1.1, L1.2	**Source:** `getLeaderboard`. Pre-seed DB with scores 100, 200, 150. Call `GET /leaderboard` with no query parameters.	Response contains entries ordered 200, 150, 100; each entry has correct `rank`, `name`, `level`, `lines`, and `score`.
WB–002	L1.1, L1.2	**Source:** `getLeaderboard`. With same dataset, call `GET /leaderboard?limit=1&offset=1`.	Response contains exactly one entry (second-best score) with `rank = 2`.
WB–003	L2.1, L2.2	**Source:** `basicTamperCheck`, `addToLeaderboard`. Compute valid HMAC `sec` for a payload; send `POST /leaderboard` with matching data and `sec`.	Request returns success; new row is inserted into database with the given fields and the same `sec`.
WB–004	L2.1, L2.2	**Source:** `basicTamperCheck`. Send `POST /leaderboard` with a random or incorrect `sec`.	Request returns HTTP 403; no new row is inserted into the database.
WB–005	L2.3	**Source:** `addToLeaderboard`. Send a valid `POST /leaderboard` with some `sec`; repeat the request with the same `sec`.	First request succeeds; second request returns a conflict-style error and the database still has only one row with that `sec`.
WB–006	L2.4	**Source:** startup / lifespan logic. Attempt to start the app without the required environment secret.	Application startup fails with a clear error message; server does not run in an insecure state.

2. Testing and Results

Because a Tetris-style game has many interacting systems (spawning, movement, collisions, line clears, scoring, levels, and backend leaderboards), I spent extra time designing test cases to cover as many situations as possible. Table 1 summarizes the core automated test cases from Section 1 and records whether they passed in the final build.

2.1 Basic Sanity Tests

The basic sanity tests were quick checks to make sure the core game loop and backend service were working at all before running longer suites. On the frontend side, I focused on BB–001 through BB–005:

- BB–001 and BB–002 confirmed that starting the game and spawning new pieces worked as expected.
- BB–003 and BB–004 verified that the horizontal movement and basic falling controls behaved correctly and stayed within the board.
- BB–005 checked that the rotations did not cause the pieces to overlap the walls or stack.

On the backend side, WB-001 was run to ensure that the leaderboard endpoint returned well-formed, sorted data from a seeded database. All basic sanity tests passed, which gave enough confidence to move on to broader smoke tests.

2.2 Smoke Tests

Smoke tests were short run-by-run runs that exercised a full "happy path" through the system without trying to hit every edge case. A typical smoke run included:

- Starting a game (BB–001),
- Playing long enough to clear at least one line (BB–006),
- Letting the grid fill to trigger the game over and show the final score (BB–010),
- Submitting the score and viewing the leaderboard (BB–014, WB–001).

These smoke tests were run after major merges or changes to the game logic or leaderboard API. The goal was simply to answer "Does a player still get a playable game, a final score, and a visible leaderboard entry?" All smoke tests passed in the final version.

2.3 Regression Tests

Regression tests were re-runs of the full automated suite whenever I changed something that could affect existing behavior (for example, tweaks to gravity, scoring, level thresholds, or leaderboard security).

For regression, I ran again:

- All UI tests BB–001 through BB–014 (spawning, controls, collision, line clears, scoring, levels, game over, start/pause/reset, and leaderboard via UI).
- All API tests WB–001 through WB–006 (ordering, pagination, signature checks, duplicate protection, and secret-enforced startup).

Each run was recorded with a date, build tag, and a list of tests, along with any failures and links to issues, following the approach described in Section 1.6. In the final build, all regression runs were completed with PASS for all test cases. Minor behavior notes, such as slightly "stiff" rotation in some wall situations, are captured as known issues (KB–01) rather than failing tests because they do not violate the current written requirements.

2.4 Integration Tests

Integration tests focused on how the React frontend and FastAPI backend worked together, rather than testing each piece in isolation.

Key integration scenarios included:

- **End-to-end score submission:** Using BB–014 on the UI side together with WB–003 on the API side to verify that a real game session could submit a signed score and immediately see it on the leaderboard.
- **Leaderboard paging from the UI:** Verification that the UI requests for different leaderboard pages were correctly mapped to backend pagination (WB–002).
- **Security and startup:** Confirming that the frontend could not accidentally talk to a backend started without a secret (WB–006) and that tampered scores were rejected both at the HTTP level (WB–004) and in the UI (error feedback).

All integration tests passed. Any unexpected behavior seen during integrated play (for example, slightly jerky gameplay on very low-end hardware at high levels) is documented as KB–02 rather than as a strict integration failure.

2.5 Performance Tests

Full load and stress testing were outside the scope of this project, but I still performed some light performance checks to make sure the game felt responsive under typical use.

Informal performance tests included:

- Play multiple games back-to-back through higher levels while watching for input lag or frame drops (related to BB–009, BB–010, BB–011).
- Submitting several scores in a row and refreshing the leaderboard many times (WB–001, WB–002, WB–003, WB–005) to ensure that the API remained responsive

On a normal development machine, the game loop stayed responsive during normal play, and leaderboard calls remained fast. On very low-end machines at very high levels, I observed occasional minor frame drops; this is captured as KB–02 in the Known Bugs section. There were no crashes or data corruption issues during these performance-oriented runs.

3. Known Bugs and Issues

At the time of writing, we track any remaining bugs or limitations in a small table. If there are no known bugs, we can state that explicitly. Otherwise, each row provides an ID, area, description, impact, and any workaround.

If by submission time all known issues are fixed, we can instead state:

> At this time, there are no known open bugs. All released functionality passes the tests listed in Section 1.

Test #	Purpose / Area	Expected	Result
BB–001	Start game spawns first tetromino (G1.1, G8.1).	After clicking Start, a tetromino appears in the top spawn row and the game state is running.	PASS
BB–002	New piece spawns only after previous one locks (G1.2, G3.1).	After the first piece reaches the bottom and locks into the stack, a new tetromino appears at the top.	PASS
BB–003	Left and right movement and wall boundaries (G2.1, G3.2).	Piece moves left or right until it reaches the board edge; extra key presses do not move it out of bounds.	PASS
BB–004	Soft-drop and hard-drop behavior (G2.2, G3.1).	Soft-drop speeds up falling in steps; hard-drop moves the piece to the lowest valid position and locks it.	PASS
BB–005	Rotation near walls and stack (G2.3, G3.1, G3.3).	Rotations that would overlap walls or existing blocks are blocked or adjusted so no overlapping blocks appear.	PASS
BB–006	Single line clear logic (G4.1, G3.1).	Exactly one full row disappears; all rows above move down by one and the cleared row becomes empty.	PASS
BB–007	Multi-line clear and gravity (G4.2, G4.3).	Two or more completed lines are removed together; score and line count match a multi-line clear; blocks above fall correctly.	PASS
BB–008	Gravity after partial clears (G4.3).	After a lower line is cleared, blocks above fall straight down with no diagonal shifts or floating gaps.	PASS
BB–009	Scoring for single vs multi-line clears (G5.1, G5.2).	Score increases after each clear; multi-line clears award more points than single-line clears; score display updates immediately.	PASS
BB–010	Game over screen and final score (G5.3, G7.1, G7.2).	When the grid fills and a new piece cannot spawn, the game stops and a final score is shown that matches the last in-game score.	PASS
BB–011	Level progression and increased fall speed (G6.1, G6.2).	After clearing the configured number of lines, the level indicator increases and the piece fall interval becomes shorter.	PASS
BB–012	Pause and resume behavior (G8.1, G8.2, G8.3).	While paused, the piece does not move and controls do nothing; after resume, falling continues from the same position and controls work again.	PASS
BB–013	Full reset behavior (G8.4, G5.2, G6.1).	Board is cleared; score, lines, and level are reset to starting values; the game is ready for a new run.	PASS
BB–014	End-to-end leaderboard submission (G5.4, G5.5, L1.1, L2.1).	After game over and score submission, the leaderboard shows the new entry in the correct sorted order.	PASS
WB–001	Basic leaderboard ordering (L1.1, L1.2).	GET /leaderboard returns scores ordered from highest to lowest with correct rank, name, level, lines, and score.	PASS

Table 1: Summary of executed automated test cases for Testrist

ID	Area	Description	Impact	Workaround / Notes
KB–01	Gameplay	Occasionally, rotation near the stack can feel “stiff” and may not wall-kick as expected for certain shapes.	Low	Player can move one cell away from the wall before rotating. Future work could refine the rotation system.
KB–02	Performance	On very low-end machines, very high levels (fast gravity) can cause minor frame drops.	Medium	Closing other heavy browser tabs usually fixes the issue. Performance tuning is possible in later versions.