JPE Sims 4 Translation Suite — Steam Deck Edition

Standalone Steam Deck Application PRD
(With Predictive Coding System + File System Skeleton)

0. Document Purpose

This document defines the full Product Requirements for the
Steam Deck edition of the JPE Sims 4 Translation Suite
(“JPE Studio: Deck Edition”) and includes:

- A complete, high-detail PRD tailored to Steam Deck.
- A controller-driven predictive coding and modding system.
- A concrete file system skeleton for implementation.

No sections are left as placeholders; everything here is
intended to be directly buildable by an engineering team.

1. Product Overview

1.1 Purpose

The goal is to build a native Steam Deck application,
“JPE Studio: Deck Edition”, that:

- Runs cleanly in both SteamOS Game Mode and Desktop Mode.
- Uses the same core translation engine and language support
  as the desktop JPE Sims 4 Translation Suite:
  - Reads Sims 4 mod formats:
    - XML tuning
    - STBL
    - .package containers
    - .ts4script (Python script archives)
    - Raw .py scripts
    - JSON, cfg, ini configuration files
  - Translates mods into Just Plain English (JPE).
  - Translates mods into JPE-XML (English-first XML fork).
  - Compiles JPE / JPE-XML back into valid Sims 4 XML
    tuning and STBL files.
- Adds a controller-driven predictive coding and modding
  system that lets users:
  - Browse projects and files.
  - Edit JPE / JPE-XML / XML content.
  - Accept, reject, and cycle through predictive
    suggestions.
  - Apply templates, macros, and quick-fixes.

  - Run validation and apply diagnostics-driven
    corrections.

The result is a “couch-IDE” for Sims 4 modding: a tool that can be fully driven from the Steam Deck controller while preserving the power of the desktop JPE Suite.

1.2 Target Users

Primary audiences:

- Sims 4 modders who:
  - Use a Steam Deck as a primary or secondary development machine.
  - Prefer controller-based or mixed controller/touch workflows.
  - Want to quickly prototype, translate, and adjust mods without a full desktop workstation.

- Existing JPE Suite users who:
  - Already use the desktop app on Windows/Linux.
  - Want a “paired” Deck environment for quick edits, field debugging, and predictive refactors while away from their main setup.

1.3 Non-Goals

This project does NOT aim to:

- Run The Sims 4 itself as part of the app.
- Turn the Deck into a generic IDE for every language (focus remains on JPE, JPE-XML, Sims 4 XML and related configs/scripts).
- Provide online multiplayer features or game-like meta systems beyond mod authoring, inspection, and diagnostics.
- Implement cloud sync in v1.0 (future extension only).

2. Platform & Runtime

2.1 Target Hardware & OS

- Primary hardware:
  - Steam Deck LCD model.
  - Steam Deck OLED model.
- Operating system:
  - SteamOS 3.x (Arch-based).
- Display:
  - Native resolution: 1280×800 @ 60Hz.

  - Must render correctly when scaled to 1152×720 or
    upscaled to 1920×1080 on external displays.
- Input devices:
  - Deck controller (sticks, D-pad, ABXY, bumpers,
    triggers, L4/L5/R4/R5, trackpads).
  - On-screen keyboard.
  - Touchscreen.
  - Optional: Bluetooth/USB keyboard and mouse in
    Desktop Mode.

2.2 App Type & Packaging

- Application type:
  - Standalone desktop-style app for SteamOS.
- Primary distribution artifacts:
  - Self-contained AppImage:
    - jpe-studio-deck-<version>.AppImage
  - Optional Flatpak manifest + build:
    - org.jpe.studio.deck
- Steam integration:
  - AppImage is added as a Non-Steam Game entry.
  - Include a desktop file:
    - Name: JPE Studio: Deck Edition
    - Exec: jpe-studio-deck.AppImage
    - Categories: Development;Utility;
    - X-SteamDeck-GamepadUI-Layout: controller
  - Must support Steam Input configuration (showing Deck
    glyphs and a default layout).

2.3 Game Mode vs Desktop Mode

Game Mode:
- Fullscreen, single-window experience.
- Controller-first navigation and interactions.
- Simplified top-level navigation:
  - Projects
  - Files
  - Editor
  - Problems
  - Predictive
- Uses Steam Input's "Desktop" or custom layout profile,
  but all major workflows must be reachable via buttons.

Desktop Mode:
- Runs as a resizable window.
- Supports keyboard, mouse, and controller simultaneously.
- Additional affordances:
  - Multi-pane layouts (e.g., Files | Editor | Problems).
  - Menubar with keyboard shortcuts.
  - Right-click context menus.

  - Denser UI components where screen real estate allows.

```
=====================================================
3. Functional Overview
=====================================================
```

3.1 Core Features (Parity with Desktop)

----

1) Project Management
- Open/close projects mapped to standard directories.
- Scan a project for mod-relevant file types:
  - .package, .ts4script, .py, .xml, .stbl, .json, .cfg,
    .ini, plus JPE/JPE-XML extensions.
- Build and maintain an index of:
  - Files.
  - Known Sims 4 tuning records.
  - JPE entities (interactions, buffs, traits, etc.).
- Provide project-level settings:
  - Game path mapping (where the player keeps Sims 4).
  - Build output path configuration.
  - Language / locale preferences.

2) Translation Engine Integration
- One-tap “Translate to JPE” flow for:
  - Individual XML tuning files.
  - Groups of files selected from the file tree.
- One-tap “Generate JPE-XML” from existing XML tuning:
  - JPE-XML is the English-first XML fork used by the
    JPE Suite.
- Round-trip compilation:
  - JPE → XML tuning + STBL (where text is moved to
    string tables).
  - JPE-XML → XML tuning + STBL.
- Ensure each round-trip is IR-driven:
  - Tunings are transformed to and from an internal
    IR (Intermediate Representation) to guarantee
    consistency and minimize format drift.

3) Diagnostics & Error Reports
- The engine runs validators during translation
  and compilation, reporting:
  - Structural violations (invalid elements, missing
    attributes, schema mismatches).
  - Semantic issues (invalid references, out-of-range
    values, mis-typed enums, missing resources).
- Errors and warnings are listed in a “Problems” panel
  with fields:
  - Severity (Error / Warning / Info).
  - File path.

  - Line / column (if applicable).
  - Error code.
  - Description.
- Exportable reports:
  - Human-readable summary (Markdown or HTML).
  - Machine-readable JSON suitable for integration
    with other tools.

4) View Modes
- JPE View (plain English DSL):
  - Focused on readability and simple editing using
    English-style syntax.
- JPE-XML View (English-friendly XML fork):
  - Exposes XML structure with more human-friendly
    tag names and attributes.
- Raw XML View:
  - Shows the original tuning exactly as found or
    generated.
  - Can be marked read-only or guarded by confirmation
    prompts for edits.
- Diff View:
  - Side-by-side or inline diff between:
    - Original vs. translated.
    - Pre-edit vs. current version.

3.2 Steam Deck-Specific Features

- Controller-centric navigation of the entire app:
  - No workflow is allowed to be “mouse only”.
- Predictive coding overlay triggered by buttons and
  contextual events.
- Layout scaling tuned for a 7" 800p screen:
  - Default editor font size: ~13–14 pt equivalent.
  - Minimum interactive target size: 40 px.
- Minimal text density in Game Mode, with more detail
  available via overlays and dedicated panels.

===
4. Architecture
===

4.1 High-Level Components
----

1) UI Shell (Deck Frontend)
- Implemented using a cross-platform UI toolkit such as:
  - Tauri (Rust backend + web frontend), or
  - Qt6/QML, or equivalent.
- Responsibilities:
  - Render the main window or fullscreen surface.

  - Manage navigation between top-level views.
  - Provide editor surfaces for JPE, JPE-XML, and XML.
  - Display overlays (predictive actions, radial menus, quick-fix choice dialogs).
  - Adapt layouts to Game Mode vs Desktop Mode.

2) Core Engine (Shared Library)
- Shared between desktop and Deck builds, responsible for:
  - Parsing JPE and JPE-XML.
  - Parsing Sims 4 XML tuning.
  - Generating XML tuning and STBL from IR.
  - Building a normalized IR representation of all tuning entities.
  - Running validation and optimizing diagnostics.

3) Predictive Engine
- Local component that provides predictive suggestions grounded in:
  - Token-level patterns (n-grams).
  - IR-context (interaction, buff, trait, etc.).
  - User history and usage statistics.
- Backed by a local SQLite database that stores:
  - Token statistics.
  - Reusable templates.
  - User macros.
  - Ranking signals for suggestions.
- Public API examples:
  - suggest_next_token(context_window).
  - suggest_template(context_type, mod_domain).
  - rank_suggestions(user_history, global_patterns).
  - record_applied_suggestion(suggestion_id, context).

4) File I/O and Mod Container Handlers
- Adapter layer for file types:
  - package_adapter: introspects .package containers.
  - stbl_adapter: reads and writes STBL entries.
  - xml_adapter: loads XML tuning as DOM/IR.
  - ts4script_adapter: enumerates and optionally reads Python scripts in .ts4script archives.
- All adapters feed data into the core engine’s IR so that predictive systems and diagnostics operate on consistent structures.

5) Diagnostics Service
- Central diagnostic bus and cache.
- Collects error/warning/info from:
  - Core engine validators.
  - Adapters.
  - Predictive engine (e.g., “this pattern is unusual”).
- Provides both:

  - Live updates for UI.
  - Cached view for Problems panel and quick-fix
    features.

4.2 Process Model

- Single OS process:
  - Main UI thread.
  - Background worker threads for:
    - Project file scanning.
    - Translation and compilation tasks.
    - Predictive suggestion generation and ranking.
- All heavy operations are cancellable:
  - Controller mapping to cancel/abort long tasks.
  - Progress indicators when operations take more
    than a short threshold.

5. UX in SteamOS Game Mode

5.1 Navigation Model

Game Mode UX is built around top-level tabs and controller
input:

- Top-level tabs (accessible via L1/R1):
  - Projects
  - Files
  - Editor
  - Problems
  - Predictive

Controls:
- Left joystick:
  - Vertical navigation in lists and tree views.
- Right joystick:
  - Horizontal panning in editor when needed.
- Right trackpad:
  - Mouse-like pointer emulation for precise actions.
- Touchscreen:
  - Optional; tap to open files, apply suggestions,
    or activate controls.

5.2 Screen Layouts

Projects Screen:
- Vertical list with each entry showing:
  - Project name.

  - Last opened time.
  - Number of mods detected.
  - Count of current errors.
- Actions:
  - A: Open project.
  - X: Open project settings.
  - Y: Rescan mods in the project.
  - Start/Menu: Add/Remove project.

Files Screen:
- Tree or filtered list of files in project.
- Filtering modes:
  - All files.
  - Only XML tuning.
  - Only JPE/JPE-XML.
  - Only files with diagnostics.
- Controller actions:
  - A: Open file in editor.
  - X: Toggle filter.
  - Y: Quick actions (Translate, Compile, Show in Problems).

Editor Screen:
- Central editor region (for JPE / JPE-XML / XML).
- Top strip:
  - File name.
  - View mode toggle (JPE / JPE-XML / XML / Diff).
  - Translate / Compile buttons.
- Bottom strip:
  - Current line/column.
  - Current input mode (Insert/Selection).
  - Suggestion state indicator (e.g., suggestion
    available, none, or quick-fix ready).

Problems Screen:
- List of diagnostics sorted by severity and file.
- Shows:
  - Severity icon.
  - Short description.
  - File path and line.
- Controller actions:
  - A: Jump to line in editor.
  - Y: Filter by severity (Errors / Warnings / Info).
  - R4: Open quick-fix suggestions (integration with
    predictive engine).

Predictive Screen:
- Grid or tiled layout of:
  - Top recommendation templates.
  - Recently applied actions.
  - User-defined macros.

- Controller actions:
  - Navigate tiles with joystick or D-pad.
  - A: Apply selected template/macro to current file and location (with confirmation if needed).

6. UX in SteamOS Desktop Mode

- Windowed experience with optional fullscreen.
- Multi-pane layout:
  - Left: Files tree.
  - Center: Editor.
  - Bottom or right: Problems and Predictive panels.
- Menubar:
  - File:
    - New, Open, Close, Save, Build, Exit.
  - Edit:
    - Undo, Redo, Cut, Copy, Paste, Find, Replace.
  - View:
    - Toggle panels, switch themes, zoom controls.
  - Tools:
    - Translation, Compilation, Validation, Reports.
  - Predictive:
    - Manage templates, macros, training, debug view.
  - Help:
    - About, documentation, controller reference.
- Keyboard shortcuts:
  - Ctrl+P: Quick file open dialog.
  - Ctrl+Space: Trigger predictive suggestions at caret.
  - Ctrl+Shift+T: Insert tuning template chooser.
  - Ctrl+Shift+E: Show Problems panel.
  - Ctrl+Shift+M: Show Macros manager.
- Controller mappings still apply so users can seamlessly switch between keyboard/mouse and Deck input.

7. Predictive Coding & Modding System

7.1 Goals

- Enable mod creation and editing with minimal typing, heavily leveraging predictions, templates, and macros.
- Map predictive control to the Steam Deck controller so that all major operations (accept suggestion, cycle options, apply templates, run quick-fixes) can be done without a physical keyboard.
- Use Sims 4–specific structure (interactions, buffs, traits, loot, autonomy tuning, etc.) to provide

  context-aware suggestions instead of generic text
  autocomplete.

7.2 Data & Models (SQLite)

The predictive engine uses a local SQLite database with
structured metadata.

Example tables:

token_stats:
- token TEXT
- context_hash TEXT
- frequency INTEGER
- last_used_ts INTEGER

templates:
- template_id TEXT PRIMARY KEY
- name TEXT
- category TEXT
  - e.g., "interaction", "buff", "trait",
        "loot_action", "career", etc.
- content_jpe TEXT
- content_jpe_xml TEXT
- usage_count INTEGER

user_macros:
- macro_id TEXT PRIMARY KEY
- label TEXT
- script TEXT
  - JPE snippet or transformation specification.
- bound_button_combo TEXT
  - e.g., "L4+Y" or "R4+X".

All data is stored locally under the user's profile
directory and never exfiltrated by default.

7.3 Controller Layout for Predictive Editing

Controller mappings specifically targeting predictive
editing flows in Game Mode:

- A:
  - Accept current suggestion.
  - If a template or macro is highlighted, apply it.

- B:
  - Dismiss current suggestion or predictive overlay.
  - Cancel out of quick-fix dialogs.

- X:
  - Cycle to the next suggestion candidate in a group.
  - When a bottom-row suggestion bar is present,
    move to the next chip.

- Y:
  - Open "Predictive Actions" overlay for current context.
  - Acts as the main entry to predictive operations
    from the editor.

- L1 / R1:
  - Switch predictive category within the overlay:
    - Tokens
    - Templates
    - Refactors (quick-fixes)
    - Macros

- L2:
  - Step backward in suggestion history at this location.
  - Useful for cycling back through previously accepted
    suggestions if the user changes their mind.

- R2:
  - Step forward in suggestion history (redo).

- L3 (click):
  - Toggle "Prediction Lock":
    - Keeps the suggestion overlay visible even when
      the cursor moves slightly.

- R3 (click):
  - Trigger "Quick Suggest":
    - One-shot suggestion at the caret based on local
      context without opening the full overlay.

- D-pad Up/Down:
  - Navigate vertical suggestion lists in the overlay.

- D-pad Left/Right:
  - Insert next/previous token suggestion inline in
    the editor when a token suggestion sequence is
    active.

- L4 (back paddle):
  - Execute the highest-ranked user macro that matches
    the current IR context.

- L5 (back paddle):
  - Open the user macro picker overlay (radial or list).

- R4 (back paddle):
  - Apply the recommended quick-fix for the current diagnostic at the caret (if available).

- R5 (back paddle):
  - Open a "Recent Templates" overlay, showing the most used templates in the current project.

- Right trackpad:
  - Fine-grained cursor positioning and selection when dealing with text.

7.4 Predictive Workflows

A) Predictive Authoring of a New Interaction

1) The user opens a blank JPE file in the editor.
2) They press Y to open the Predictive Actions overlay.
3) They navigate to the "Templates" category and select "Interaction" using the joystick/D-pad and A.
4) The engine inserts a full JPE interaction skeleton, including placeholders for:
   - Title.
   - Actor/Target roles.
   - Tests/conditions.
   - Loot actions.
   - Tooltip or strings.
5) The cursor jumps to the title field. Inline suggestions appear, e.g., title variations derived from:
   - File name.
   - Similar interactions in the project.
   - Frequently used patterns.
6) The user cycles suggestions with D-pad Left/Right and accepts one with A.
7) As the user moves through the skeleton, the engine offers context-aware suggestions for:
   - tests (e.g., sim traits, skills, moods).
   - loot actions.
   - buff references.
   - autonomy settings.

B) Predictive Refactor for Existing Mod

1) The user opens a JPE file containing existing content.
2) Diagnostics show issues (e.g., "Buff reference not found").
3) In the Problems panel, the user highlights the error and presses R4 to request quick-fixes.
4) The predictive engine identifies likely options:

   - Create a new Buff template with the referenced ID.
   - Replace with an existing Buff from the project index.
5) A small overlay provides options; the user selects with
   D-pad and presses A.
6) The engine applies the refactor, updating references and
   optionally inserting new JPE Buff definitions.

C) Quick-Fix Loop
----
1) The user triggers "Compile" (e.g., via Start → Compile).
2) Diagnostics populate the Problems panel.
3) The user navigates the errors using the joystick and
   highlights the top error.
4) Pressing R4 opens predicted quick-fixes for that error:
   - Fill missing attribute with a default.
   - Wrap value in required container element.
   - Generate referenced asset (buff/trait/loot).
5) The user chooses a fix and applies it with A.
6) The predictive system records which fixes are most
   frequently chosen, increasing their prominence in future
   similar contexts.

7.5 Predictive Overlays & Visual Elements

Inline Suggestion Chips:
- Appear directly near the caret.
- Simple rectangular or pill-shaped UI elements.
- A: accept.
- B: dismiss.
- Designed to be visually unobtrusive but clearly legible.

Bottom Bar Suggestion Row:
- At the bottom edge of the editor.
- Up to 3 suggestion “chips” such as:
  - [Add condition]
  - [Add loot action]
  - [Add tooltip]
- X/Y or D-pad can move between chips; A applies the
  currently highlighted suggestion.

Radial Menu for Macros & Templates:
- Triggered by holding L5 for ~0.5 seconds.
- Radial sectors for:
  - User macros.
  - Interaction templates.
  - Buff templates.
  - Trait templates.
  - Quick-fix bundles.
- User moves the left joystick to select a sector and

releases L5 to apply.

7.6 Integration with Diagnostics & IR

- Each suggestion is annotated with metadata:
  - source_type:
    - ngram
    - template
    - macro
    - quick_fix
  - confidence_score (0.0 to 1.0).
  - ir_context (e.g., "interaction.test_set",
    "loot", "buff", "trait").
- When frequent error patterns are detected, the predictive
  engine promotes useful quick-fixes into first-class
  templates and makes them easier to access in similar
  contexts.

==
8. File Handling & Mod Integration
==

8.1 Supported File Types

Read/Write:
- JPE (.jpe or .jpe.txt).
- JPE-XML (.jpe.xml).
- Raw XML tuning.
- JSON configuration files used by mods.

Read-Only (via adapters):
- .package containers (tuning, resources).
- .ts4script (Python archives).
- STBL string tables.
- cfg and ini configuration files.

8.2 Safe Write Strategy

- Original mod files are never overwritten directly.
- All tool-generated files, translations, and compilations
  are written into dedicated build directories inside the
  project:
  - jpe_build/
  - dist/
- Each build operation emits a build report that includes:
  - Files generated.
  - Errors and warnings encountered.
  - Summary of transformations (JPE → XML, etc.).

8.3 Offline Operation

```
--------------------
- All components (core engine, predictive system, adapters)
  operate fully offline.
- No network connectivity is required to function.
- Optional future enhancements might introduce cloud sync
  or shared predictive models, but v1.0 is entirely
  local-first.

======================================================
9. Performance & Constraints
======================================================

- Cold start time:
  - Under 5 seconds on a Steam Deck SSD for a moderate
    project size.
- Project scanning:
  - Capable of indexing ~5,000 files without blocking the
    UI by using worker threads and progress UI.
- Predictive latency:
  - Initial suggestion must appear within 150 ms of a
    triggering event (cursor movement, text edit).
- Memory usage:
  - Target under 1.5 GB RAM for a typical session with:
    - One medium project loaded.
    - Predictive engine active.
    - Diagnostics open.

======================================================
10. Security & Privacy
======================================================

- All processing happens on user-owned mods and files.
- By default, there is no telemetry.
- Local data (SQLite DB, settings, caches) is stored under:
  - $HOME/.local/share/jpe-studio-deck/
- Optional setting:
  - “Allow anonymous usage metrics” (off by default).
  - If enabled, metrics must never include raw mod content
    or identifiable user information.

======================================================
11. Testing & QA
======================================================

11.1 Core Engine Regression
--------------------------
- Use the same unit and integration tests as the desktop
  engine where applicable:
  - XML → JPE → XML equivalence tests.
  - XML → JPE-XML → XML equivalence tests.
```

  - Diagnostics tests on curated “bad mod” fixtures.

11.2 UI & Controller Tests

- Manual scripts for:
  - Navigating all screens using controller only.
  - End-to-end workflow tests:
    - Opening a project.
    - Translating a file.
    - Editing in JPE.
    - Running compile.
    - Applying quick-fixes via controller.
  - Predictive overlay responsiveness tests.

- Automated tests (depending on toolkit):
  - Component-level tests for:
    - Editor.
    - Problems panel.
    - Predictive overlays.
    - Radial menu controls.

11.3 Performance & Stress Tests

- Heavy project scenario:
  - At least 1,000 XML tuning files.
  - At least 50 JPE files.
- Measurements:
  - Startup time.
  - Index build time.
  - Average predictive response time for frequent actions.

==
12. Release & Distribution
==

12.1 Versioning

- engine_version:
  - Shared with desktop JPE Suite core engine.
- jpe_version:
  - Version of the JPE language supported.
- jpe_xml_version:
  - Version of the JPE-XML schema supported.
- deck_app_version:
  - Semantic version (e.g., 1.0.0, 1.1.0).
- predictive_schema_version:
  - Schema version of the predictive SQLite layout.

12.2 Release Artifacts

- jpe-studio-deck-<deck_app_version>.AppImage
- jpe-studio-deck-flatpak.json (and built Flatpak if used).
- Release notes documenting:
  - New features.
  - Fixed defects.
  - Known issues.
  - Platform / compatibility notes.

12.3 Installation Flow (User-Facing)

1) Download the AppImage onto Steam Deck.
2) From Desktop Mode:
   - Mark it executable:
     - chmod +x jpe-studio-deck-1.0.0.AppImage
3) Add it to Steam as a Non-Steam Game entry.
4) Optionally configure a custom Steam Input profile
   for the best controller layout.
5) (Optional) Install Flatpak via Discover if Flatpak
   packaging is provided.

13. Appendix A — File System Skeleton

Project root layout for the Steam Deck app implementation:

```
jpe-studio-deck/
■■■ README.md
■■■ LICENSE
■■■ deck_app/
■   ■■■ __init__.py
■   ■■■ main.py
■   ■■■ config/
■   ■   ■■■ __init__.py
■   ■   ■■■ settings.yaml
■   ■■■ ui/
■   ■   ■■■ __init__.py
■   ■   ■■■ shell.py
■   ■   ■■■ editor.py
■   ■   ■■■ predictive_overlay.py
■   ■   ■■■ problems_panel.py
■   ■   ■■■ controllers.py
■   ■■■ core/
■   ■   ■■■ __init__.py
■   ■   ■■■ engine_adapter.py
■   ■   ■■■ file_indexer.py
■   ■   ■■■ diagnostics_bridge.py
■   ■   ■■■ predictive_engine.py
■   ■   ■■■ sqlite_store.py
■   ■■■ platforms/
■   ■   ■■■ steamdeck/
■   ■       ■■■ __init__.py
■   ■       ■■■ game_mode.py
■   ■       ■■■ desktop_mode.py
■   ■       ■■■ input_mapping.py
■   ■■■ assets/
■       ■■■ icons/
■       ■   ■■■ app_icon.png
■       ■   ■■■ tray_icon.png
■       ■■■ themes/
■           ■■■ dark_deck.json
■           ■■■ high_contrast.json
■■■ core_engine/
■   ■■■ __init__.py
■   ■■■ translator.py
■   ■■■ jpe_parser.py
■   ■■■ jpe_xml_parser.py
■   ■■■ xml_generator.py
■   ■■■ stbl_handler.py
■   ■■■ diagnostics.py
■■■ predictive/
■   ■■■ __init__.py
■   ■■■ models.py
■   ■■■ ngram_store.py
■   ■■■ templates_registry.py
■   ■■■ macros.py
■   ■■■ ranking.py
■■■ adapters/
■   ■■■ __init__.py
■   ■■■ package_adapter.py
■   ■■■ stbl_adapter.py
■   ■■■ xml_adapter.py
■   ■■■ ts4script_adapter.py
■■■ tests/
■   ■■■ __init__.py
■   ■■■ test_core_engine.py
■   ■■■ test_predictive_engine.py
■   ■■■ test_steamdeck_input_mapping.py
■   ■■■ test_file_indexer.py
■   ■■■ fixtures/
■       ■■■ sample_tunings/
■       ■■■ sample_jpe/
■       ■■■ sample_jpe_xml/
■■■ packaging/
■   ■■■ appimage/
■   ■   ■■■ build.sh
■   ■   ■■■ AppRun
■   ■■■ flatpak/
■   ■   ■■■ manifest.json
■   ■   ■■■ org.jpe.studio.deck.desktop
■   ■■■ steam/
■       ■■■ jpe-studio-deck.desktop
■■■ docs/
    ■■■ deck_prd.md
    ■■■ controller_mappings.md
    ■■■ predictive_engine_design.md
```

==

End of Document

==