JPE Predictive Scripting & Coding Module PRD
Aligned with SOP — JPE Sims 4 Mod Translation Suite

1. Purpose & Scope

1.1 Purpose

The JPE Predictive Scripting & Coding Module ("Predictive Module") extends the JPE Sims 4 Mod Translation Suite by providing intelligent, offline-first assistance when authoring, editing, and debugging mods in Just Plain English (JPE) and JPE-XML. It uses a local SQLite database to store patterns, snippets, diagnostics, and usage statistics so the tool can "learn" from existing mods and user behavior without requiring a network connection.

The module augments, but does not replace, the core translation pipeline defined in the SOP:
- Existing XML → IR → JPE / JPE-XML
- JPE / JPE-XML → IR → XML tuning and STBL

The Predictive Module operates on top of the IR and language definitions to suggest:
- Next lines and blocks of JPE/JPE-XML as you type.
- Canonical patterns for common Sims 4 interactions, buffs, traits, and loot.
- Quick-fix actions for diagnostics and validation errors.
- Reusable snippet libraries for frequently used tuning structures.

1.2 Scope

In scope for this PRD:
- A predictive authoring engine for JPE and JPE-XML.
- A complete SQLite-backed pattern/snippet/diagnostics library.
- Integration with desktop JPE Studio and the iOS app.
- APIs for core/ and plugins/ to query or extend predictive behavior.
- Offline-first operation, with optional export/import for knowledge sharing.

Out of scope for this PRD (future work):
- Cloud-scale collaborative learning service.
- Integration with third-party online LLM APIs.
- General-purpose AI code generation for non-Sims projects.

1.3 Alignment with Master SOP

This module is an extension of:
- Phase 1 — Language and IR Design: uses the IR as the canonical representation to derive patterns and canonical forms.
- Phase 2 — Core Engine Implementation: hooks into parsers, generators, and validators to collect features and suggest fixes.
- Phase 3 & 4 — Desktop / iPhone Apps: integrates into JPE Studio and the mobile client as inline suggestions, snippet palettes, and quick-fix menus.
- Section 7 — Diagnostics and Exception Handling: uses diagnostic output as a primary driver for fix suggestions and guided editing.

2. Goals, Non-Goals & Success Criteria

2.1 Goals
- Make writing JPE/JPE-XML feel guided, with guardrails:
  - Reduce boilerplate by surfacing templates and snippets.
  - Lower error rates by suggesting valid structures and values.
  - Shorten debug cycles via one-click “Apply Fix” actions.
- Operate fully offline using a SQLite database and local mod corpus.
- Maintain transparency and control: user can see, edit, and export the predictive library
.

2.2 Non-Goals
- The module is not a black-box AI assistant; all patterns and snippets must be traceable to human-readable sources (mods, templates, or shipped libraries).
- It is not responsible for final correctness of tuning logic; the translator/validator st ill have final authority.
- It is not a general-purpose text editor; it enhances existing JPE editors.

2.3 Success Criteria
- New JPE users can author a working interaction/buff/trait with <50% of the keystrokes re quired without the module.
- Error rate (post-build diagnostics) is reduced by at least 30% on a sample corpus of new mods.
- 90%+ of suggestions in beta testing are rated “useful” or “acceptable” by test users.
- The predictive DB can be safely migrated between versions without data loss or corruptio n.

3. User Roles & Core Use Cases

3.1 Roles
- Mod Author (Beginner): Wants “fill-in-the-blanks” style support and safe templates.
- Mod Author (Advanced): Wants autocomplete, pattern reuse, and quick access to prior work
.
- Translator / Tech Editor: Uses predictive snippets to enforce consistent style and struc ture.
- QA / Debugger: Uses quick-fix suggestions tied to diagnostics to resolve issues rapidly.

3.2 Core Use Cases
UC-1: JPE Autocomplete While Typing
- As the user types JPE, the editor queries the Predictive Module for:
  - Inline completions (ghost text).
  - Multi-line snippet suggestions (e.g., full interaction scaffold).
- User confirms with Enter/Tab.

UC-2: Pattern-Based Wizards
- User chooses “New Interaction (Sim $\rightarrow$ Object)” or similar wizard.
- The Predictive Module returns canonical templates populated with default values and hint

s.
- Wizard writes JPE/JPE-XML stubs into the project.

UC-3: Quick-Fix from Diagnostics
- Validator raises an error (e.g., missing loot list, invalid test set).
- Predictive Module looks up known fixes and offers “Apply Fix” button in Problems pane.
- Applying fix inserts JPE snippet or patches JPE-XML at the correct location.

UC-4: Snippet Library Browsing
- User opens Snippet Library side panel.
- They filter by category (interaction, buff, trait, test, loot, autonomy, etc.) and tag.
- Snippet details show example usage and origins (mod/file/lines).

UC-5: Learning from Corpus
- Background job indexes existing mods and JPE files.
- Extracted patterns are stored in SQLite.
- Over time, suggestions become better tailored to the user’s style and commonly used acti ons.

4. Functional Requirements

4.1 Predictive Authoring Engine
- FR-1: Provide token-level autocomplete for JPE/JPE-XML based on:
  - Current prefix.
  - AST context (where within interaction/buff/test structure).
- FR-2: Support next-line and next-block suggestions for common patterns.
- FR-3: Recognize and promote canonical forms (e.g., full interaction definition with prop er tests/loot).

4.2 Snippet & Template Management
- FR-4: Store reusable snippets with metadata:
  - Category (interaction, buff, trait, loot, autonomy, etc.).
  - Language (JPE, JPE-XML, XML).
  - IR types referenced.
  - Tags (e.g., “social”, “career”, “needs”, “occult”).
- FR-5: Allow users to create, edit, clone, and delete custom snippets.
- FR-6: Distinguish “shipped” snippets from “user” snippets and prevent shipped ones from being edited in-place (must be cloned).

4.3 Diagnostics-Driven Suggestions
- FR-7: Link diagnostics codes to fix patterns in the DB.
- FR-8: For each diagnostic, the module can return 0..N potential fix actions:
  - Insert missing block.
  - Normalize a value.
  - Suggest a localized template.
- FR-9: Expose an API for validators and plugins to register new fix patterns.

4.4 Integration with JPE Studio (Desktop)
- FR-10: Provide an in-process API for the desktop UI to request suggestions on every keys troke (with debouncing) and on demand.
- FR-11: Surface suggestions via:
  - Inline ghost text.
  - Dropdown completion list.
  - Right-click “Quick fix” menu on diagnostics.
  - Snippet Library panel.
- FR-12: Provide configuration options for aggressiveness, ranking, and privacy.

4.5 Integration with iOS App
- FR-13: Expose a simplified query API optimized for mobile (e.g., over a local store shar ed via App Group or bundled DB).
- FR-14: Support touch-friendly snippet insertion, with previews before insertion.
- FR-15: Allow offline-only operation; sync of patterns/snippets is optional and user-cont rolled.

4.6 Offline-First & Data Ownership
- FR-16: All predictive data is stored locally in SQLite.
- FR-17: Provide export/import of the predictive DB (or subsets) as plain JSON or SQL for backup or sharing.
- FR-18: Allow users to reset or prune the DB.

4.7 Configuration & Tuning
- FR-19: Global and project-level settings for:
  - Enabled features (autocomplete, snippets, quick-fix).
  - Suggestion sources (built-in vs corpus-learned vs user-created).
  - Privacy controls and data retention thresholds.
- FR-20: Advanced users can tweak ranking weights (e.g., prefer shipped patterns vs local corpus).

5. Non-Functional Requirements

5.1 Performance
- Initial DB load: < 500ms on typical desktop, < 1.5s on phone.
- Suggestion queries: target < 50ms median response on desktop, < 120ms on phone.
- Background indexing: throttled to avoid noticeable UI lag; can be paused.

5.2 Reliability & Integrity
- SQLite DB must be ACID-compliant with journaling enabled.
- Corruption detection and auto-repair routines where possible (e.g., rebuild indexes).
- Regular backups/snapshots on schema migrations or mass imports.

5.3 Security & Privacy
- No automatic upload of predictive data.
- Sensitive paths (file locations, usernames) must be anonymized before export, or exports must be clearly labeled as containing local paths.

- User must explicitly opt in to any future cloud sync features (not part of this PRD).

5.4 Maintainability
- Database schema versioned with migration scripts.
- Predictive engine APIs must remain stable within a major version.
- All configuration is stored in human-readable files alongside or adjacent to projects wh
en possible.

6. Architecture Overview

6.1 High-Level Components
- Predictive Engine Core
  - In-memory engine that ingests prefix + AST context and returns ranked suggestions.
- SQLite Store
  - Encapsulates all DB operations: schema, queries, indexing jobs, migrations.
- Indexer
  - Runs over mods/JPE files, extracts snippets, patterns, and diagnostics correlations.
- Integration Adapters
  - Desktop Adapter: wires engine into JPE Studio’s editor and Problems pane.
  - iOS Adapter: provides a lightweight API over the on-device SQLite DB.
  - Core/Plugins Adapter: simple API for validators and plugins.

6.2 Data Flow (Typical Edit Operation)
1) User types or edits JPE.
2) Editor sends current prefix + cursor AST context to Predictive Engine.
3) Predictive Engine queries SQLite Store for matching patterns, snippets, and n-grams.
4) Engine ranks candidates, merges shipped + user + corpus-learned suggestions.
5) Editor displays results; user accepts or rejects.
6) Accepted suggestions cause “usage events” to be written back to the DB to improve ranki
ng.

7. SQLite Database Library Design

7.1 Overview
The predictive database is implemented as a single SQLite file per user profile, with opti
onal project-specific overlays. The DB is accessed via a library that exposes high-level o
perations (e.g., “find_suggestions_for_context”) rather than raw SQL to calling code.

7.2 Schema Versioning
- A table meta_info stores key/value metadata, including schema_version.
- Migrations are scripted and idempotent; application must refuse to run on unknown major
versions until migration completes or user explicitly opts to reset.

7.3 Tables

Table: meta_info
- key TEXT PRIMARY KEY

- value TEXT NOT NULL

Used for:
- schema_version (e.g., “1.0.0”)
- created_at, last_migrated_at
- engine_version, jpe_version, jpe_xml_version

Table: mods
- mod_id INTEGER PRIMARY KEY AUTOINCREMENT
- name TEXT NOT NULL
- author TEXT
- source_path TEXT
- game_version TEXT
- tags TEXT -- comma-separated or JSON
- created_at DATETIME
- updated_at DATETIME

Table: files
- file_id INTEGER PRIMARY KEY AUTOINCREMENT
- mod_id INTEGER REFERENCES mods(mod_id) ON DELETE CASCADE
- path TEXT NOT NULL
- file_type TEXT NOT NULL -- e.g., “JPE”, “JPE-XML”, “XML”
- hash TEXT -- content hash for deduping and change detection
- last_parsed_at DATETIME
- is_active INTEGER DEFAULT 1

Table: jpe_snippets
- snippet_id INTEGER PRIMARY KEY AUTOINCREMENT
- language TEXT NOT NULL -- “JPE”, “JPE-XML”, “XML”
- title TEXT -- human-friendly name
- content TEXT NOT NULL -- full snippet text
- ast_fingerprint TEXT -- IR-based hash for structure
- category TEXT -- “interaction”, “buff”, “trait”, “loot”, “test”, etc.
- tags TEXT -- comma-separated or JSON (e.g. “[social,occult]”)
- source_type TEXT NOT NULL -- “builtin”, “corpus”, “user”
- mod_id INTEGER -- nullable, origin mod if corpus-based
- file_id INTEGER -- nullable, origin file if corpus-based
- line_start INTEGER
- line_end INTEGER
- usage_count INTEGER DEFAULT 0
- quality_score REAL DEFAULT 0.0
- created_at DATETIME
- updated_at DATETIME

Table: completion_patterns
- pattern_id INTEGER PRIMARY KEY AUTOINCREMENT
- trigger_prefix TEXT NOT NULL -- textual prefix or serialized token sequence

- ast_context TEXT -- serialized AST/IR context (e.g., “Interaction>LootL
ist”)
- suggestion_snippet_id INTEGER REFERENCES jpe_snippets(snippet_id)
- confidence REAL DEFAULT 0.5
- accepted_count INTEGER DEFAULT 0
- rejected_count INTEGER DEFAULT 0
- last_used_at DATETIME

Table: diagnostics
- diag_id INTEGER PRIMARY KEY AUTOINCREMENT
- code TEXT NOT NULL -- e.g., “JPE001_MISSING_LOOT_LIST”
- severity TEXT NOT NULL -- “info”, “warning”, “error”
- message_template TEXT NOT NULL
- created_at DATETIME

Table: diagnostic_fixes
- fix_id INTEGER PRIMARY KEY AUTOINCREMENT
- diag_code TEXT NOT NULL -- FK to diagnostics.code (logical)
- title TEXT NOT NULL -- user-facing label
- description TEXT -- explanation of the fix
- snippet_id INTEGER REFERENCES jpe_snippets(snippet_id)
- apply_mode TEXT NOT NULL -- “insert_at_cursor”, “insert_block”, “replace_
range”
- priority INTEGER DEFAULT 0

Table: events
- event_id INTEGER PRIMARY KEY AUTOINCREMENT
- event_type TEXT NOT NULL -- e.g., “suggestion_accepted”, “suggestion_reje
cted”
- snippet_id INTEGER -- nullable
- pattern_id INTEGER -- nullable
- diag_code TEXT -- nullable
- file_id INTEGER -- nullable
- project_id TEXT -- nullable logical project key
- ts DATETIME NOT NULL
- details TEXT -- JSON blob for extra info

Table: user_settings
- key TEXT PRIMARY KEY
- value TEXT NOT NULL -- JSON-encoded

Optional Table: ngrams
- ngram_id INTEGER PRIMARY KEY AUTOINCREMENT
- n INTEGER NOT NULL -- 1, 2, 3...
- prefix TEXT NOT NULL -- textual / token prefix
- continuation TEXT NOT NULL -- suggested continuation
- frequency INTEGER NOT NULL DEFAULT 1

7.4 Indexes
- CREATE INDEX idx_files_mod_id ON files(mod_id);
- CREATE INDEX idx_snippets_category ON jpe_snippets(category);
- CREATE INDEX idx_snippets_language ON jpe_snippets(language);
- CREATE INDEX idx_snippets_ast_fp ON jpe_snippets(ast_fingerprint);
- CREATE INDEX idx_patterns_prefix ON completion_patterns(trigger_prefix);
- CREATE INDEX idx_patterns_ast_ctx ON completion_patterns(ast_context);
- CREATE INDEX idx_diagfix_diag_code ON diagnostic_fixes(diag_code);
- CREATE INDEX idx_events_type_ts ON events(event_type, ts);
- For ngrams: CREATE INDEX idx_ngrams_prefix ON ngrams(prefix);

7.5 Library Responsibilities
The SQLite library (e.g., predictive_store.py) must provide high-level operations:
- init_database(path) → connection
- get_schema_version() / migrate_schema()
- record_mod_file(mod_metadata, file_metadata)
- upsert_snippet(snippet)
- get_snippets_by_context(ast_context, category, tags)
- get_completion_candidates(prefix, ast_context, limit)
- record_suggestion_event(event_type, snippet_id, pattern_id, diag_code, file_id, project_id)
- get_fixes_for_diagnostic(diag_code)
- get_user_setting(key) / set_user_setting(key, value)
- export_database(filter_options) → JSON
- import_database(json_blob, strategy)

The calling code should never construct SQL manually; it uses the library API to ensure safety and migration compatibility.

8. Predictive Engine Logic

8.1 Inputs
- Textual prefix at cursor.
- AST/IR context node (e.g., “Inside Interaction: loot_actions list”).
- Optional diagnostic code (for quick-fix scenarios).
- User/project configuration (aggressiveness, sources enabled).

8.2 Ranking Strategy
- Base score from completion_patterns.confidence.
- Boost from jpe_snippets.usage_count and quality_score.
- Additional boosts for:
  - Matching ast_fingerprint.
  - Matching category and tags to project context.
  - Recently used snippets (recency bias).

8.3 Learning Loop
- When a suggestion is accepted:

  - Increment accepted_count for the associated completion_patterns row.
  - Increment usage_count and adjust quality_score for the snippet.
  - Log an “suggestion_accepted” event for analytics/tuning.
- When a suggestion is rejected or dismissed:
  - Increment rejected_count.
  - Optionally down-rank future suggestions with similar trigger_prefix + ast_context.

8.4 Fallbacks
- If no context-specific suggestion exists, fall back to:
  - n-gram continuation from ngrams table.
  - Category-level generic snippets.
- If DB is unavailable or corrupted:
  - Run in “degraded mode” using in-memory shipped snippets only (no writes).

9. Integration Details

9.1 Desktop (JPE Studio)
- Predictive Engine runs in the same process as the editor for low latency.
- Debounced calls: only re-query after a short delay since last keystroke (e.g., 75–150ms)
.
- UI Requirements:
  - Display top N suggestions ranked, with snippet preview.
  - Show origin (built-in, corpus, user) in tooltip.
  - Provide a way to pin favorite snippets.
- Editing Operations:
  - Suggestions insert text directly into editor buffer via existing edit APIs.
  - Quick-fix actions perform structured edits using IR/AST transforms where possible.

9.2 iOS App
- Use a slimmed-down SQLite DB, synchronized from desktop or created standalone.
- Suggestion calls are triggered by explicit user action (e.g., tapping “Suggest next bloc
k”) to avoid aggressive power use.
- Provide offline help: pattern descriptions and how-to text embedded in snippet metadata.

9.3 Core & Plugins
- Expose a minimal, language-agnostic API to register diagnostics and fixes:
  - register_diagnostic(code, severity, message_template)
  - register_fix_for_diagnostic(code, fix_spec)
- Allow plugins to insert specialized snippets for new packs, expansions, or game systems.

10. Data Lifecycle & Migrations

10.1 Initial Population
- Ship DB seed with:
  - Canonical JPE/JPE-XML snippets for core Sims 4 systems (using IR from SOP corpus).
  - Common diagnostics and their fixes.
- On first run, DB is copied to user profile and upgraded to current schema_version.

10.2 Indexing Existing Projects
- Background process:
  - Parse existing mods and JPE projects.
  - Extract snippets around interactions, buffs, traits, loot, tests, autonomy, etc.
  - Store them as corpus snippets (source_type = “corpus”).

10.3 Migrations
- Every schema change increments schema_version and includes:
  - SQL migration script.
  - Data migration logic (e.g., computing new fields from existing data).
- Failed migration:
  - Application logs error and offers user options:
    - Retry.
    - Reset DB (with backup export if possible).

11. Testing & QA

11.1 Unit Tests
- SQLite library:
  - Schema creation and migrations.
  - CRUD operations for snippets, patterns, diagnostics, events.
- Predictive Engine:
  - Ranking logic given controlled test data.
  - Fallback behavior when DB is missing or empty.

11.2 Integration Tests
- End-to-end tests simulating typing JPE/IPE-XML and validating:
  - Suggestions appear within acceptable latency.
  - Accepted suggestions produce valid JPE/JPE-XML that compiles back to XML without error
s.
- Diagnostic quick-fix tests:
  - For each diagnostic code with a fix, verify that “Apply Fix” results in fewer or no er
rors.

11.3 Performance & Load Tests
- Populate DB with a large corpus (e.g., thousands of mods).
- Measure query latency for typical operations.
- Stress-test migrations and export/import functionality.

11.4 UX Validation
- Beta tests with beginner and advanced modders:
  - Capture satisfaction scores for suggestions.
  - Gather feedback on noise vs usefulness and adjust ranking heuristics.

12. Risks & Mitigations

- Risk: Overly aggressive suggestions annoy users.

  - Mitigation: Configurable aggressiveness and easy toggles; conservative defaults.
- Risk: DB corruption (e.g., abrupt shutdown).
  - Mitigation: Journaling + regular backups; robust recovery path.
- Risk: Schema complexity leads to migration pain.
  - Mitigation: Careful versioning, automated tests, and explicit migration paths.
- Risk: Overfitting to a single user’s quirky style.
  - Mitigation: Balance shipped canonical snippets with user-learned patterns; allow user to reset or prune corpus.
- Risk: Predictive behavior diverges from upstream Sims 4 tuning changes.
  - Mitigation: Version fields for game_version and jpe/jpe_xml versions, plus update routines tied to new game patches.

13. Deliverables

- Predictive Engine Core library.
- SQLite database library and schema migrations.
- Seed database with canonical snippets and diagnostics/fixes.
- Desktop (JPE Studio) integration:
  - Inline autocomplete.
  - Snippet Library panel.
  - Diagnostics quick-fix actions.
- iOS integration:
  - Lightweight suggestion API.
  - Snippet browser and one-tap insertion.
- Documentation:
  - Developer guide for extending the predictive engine and DB schema.
  - User guide for configuring predictive features and managing the snippet library.