SoulCore Framework — Prototype Development Guide

Claude as Lead Developer | Human as Technical Assistant

Part 1: Critique Assessment

The Short Version

The critique is accurate and from someone who reads carefully. None of the issues identified should stop you from starting development. All five "critical" items in the summary are implementation-precision gaps, not design flaws — meaning they resolve naturally as you write the code. Here is how to think about each one:

Issue 1 — Missing public API for Tier0StateMachine *Verdict: Address during development, not before.* The critique is correct that `apply_proposal`, `export_session`, and `import_session` aren't listed in the spec. But these methods are extensively described in Sections 9 and 10, and their signatures appear in the test suite. A developer implementing them will not be confused. Document the full API in the code's docstrings as you write it; a separate spec section isn't necessary for a prototype.

Issue 2 — Undefined return structures in StorageBackend *Verdict: Correct, and the easiest fix you'll make. Add a comment block.* This is genuinely the most implementation-critical gap. When you implement `BTreeBackend`, you need to know exactly what `get_all_memories()` is supposed to return. The fix is one paragraph of inline documentation added to `storage.py` as you write it. Make this the first thing you write.

Issue 3 — Ambiguous checksum definition *Verdict: Legitimate. One-line fix. Do it now, in your head.* The checksum should be:

```
hashlib.sha256(
    json.dumps(
        {"variables": snap.variables, "last_updated": snap.last_updated, "entity_
        sort_keys=True
    ).encode()
```

```
).hexdigest()
```

That's the spec. It takes thirty seconds to decide and you'll never think about it again.

Issue 4 — Tokenization/stopwords not defined *Verdict: Worry about this tomorrow. Hardcode a list.* For the prototype, use this exact list and move on:

```
STOPWORDS = {"the","a","an","is","are","was","were","be","been","being",
             "have","has","had","do","does","did","will","would","could",
             "should","may","might","i","you","he","she","it","we","they",
             "what","which","who","that","this","these","those","and","or",
             "but","in","on","at","to","for","of","with","about"}
```

The retrieval algorithm works fine with this. Refine later.

Issue 5 — Minor missing details (synonym selection, memory_template fallback, recent_history_max) *Verdict: All "worry about tomorrow" items.*

- Synonym selection: `rng.choice(synonym_list)` . Obviously. Write it that way.
- `memory_template` fallback: if key is missing, skip enrichment silently. One `if` statement.
- `recent_history_max` in snapshot: the critique calls this "unusual" but it's actually useful — the receiving device knows how many turns to expect. This is a design preference, not a bug. You can ignore this specific note.

One item the critique gets wrong: The note about `_apply_voice_gate` and `generate_response_with_proposal` being "internal" in tests is a non-issue. Testing internal methods via underscore-prefixed names is completely normal in Python. Don't refactor on this account.

Bottom line: Start building. None of this needs a new spec revision. The v3.0 spec is implementable as written. Resolve each item at the moment you write the relevant code.

Part 2: Free vs. Pro — Which Plan Do You Need?

Recommendation: Get Pro ($20/month). It will pay for itself in the first week.

Here's the honest breakdown for this specific project:

Why free will frustrate you: The v3.0 spec is enormous — the document alone is

roughly 10,000 words of dense technical content. Every conversation where you paste the spec or reference it eats a significant portion of the free tier's ~15-message-per-5-hour window. A single productive coding session — "write this module, review it, fix this, now write the tests" — can consume that entire allowance before you're done with one file. You'll spend time waiting for resets instead of building.

Why Pro at $20 solves it: Pro gives you roughly 5× the message budget, priority access (no queue during peak hours), access to Opus (the stronger model, measurably better at code), and Claude Code (the terminal tool). For a project with this level of complexity — a multi-module Python framework with a custom DSL parser, abstract storage layer, and a formal test suite — you want the better model on the harder problems.

The Projects feature is the real unlock: Both free and Pro now include Projects, but Pro's higher limits make Projects actually useful for this. You'll create a Project, paste the full v3.0 spec into its Knowledge section once, and every conversation in that Project automatically has the spec in context. This eliminates the single biggest waste of tokens: re-explaining what you're building at the start of every session.

Could you make free work? Technically yes, if you're extremely disciplined — very focused sessions, one module per session, no backtracking. In practice, when you hit a wall at message 12 and need three more exchanges to finish debugging, you'll wish you had Pro. For a $20 decision on a project you're clearly investing serious time in, it's not worth optimizing around.

Part 3: Step-by-Step Development Guide

How This Works

You are the hands. Claude (in your Project) is the mind. Every step below either tells you what prompt to give Claude, or tells you exactly what to do with the code Claude produces. When in doubt, your job is: copy what Claude writes, put it in the right file, run the command, paste the error back. That's it.

Claude will write all code, all tests, all fixture files. You do not need to understand the code — you need to execute the instructions precisely.

One-Time Setup (Do This First)

Step 1 — Install Python

Check if Python is already installed. Open your terminal (on Mac: press Cmd+Space, type "Terminal", press Enter; on Windows: press the Windows key, type "cmd", press Enter) and type:

```
python --version
```

If you see a number starting with 3.8 or higher, you're fine. If you see an error or a number below 3.8, go to https://www.python.org/downloads/ and download the latest version. During installation on Windows, check the box that says "Add Python to PATH."

Step 2 — Install pytest

In the same terminal, type:

```
pip install pytest
```

Wait for it to finish. If you see "Successfully installed pytest", continue. If you see an error, paste the error into Claude and ask for help.

Step 3 — Create your project folder

In the terminal, type these commands one at a time, pressing Enter after each:

```
mkdir soulcore
cd soulcore
mkdir -p scf scf-devtools/tests/unit scf-devtools/tests/integration scf-devtools/
```

This creates the folder structure the spec describes. All future terminal commands should be run from inside the `soulcore` folder.

Step 4 — Set up your Claude Project

In Claude.ai:

1. Click "Projects" in the left sidebar
2. Click "New Project"
3. Name it "SoulCore Framework"
4. Click "Add content" or "Project knowledge"
5. Paste the entire v3.0 spec document into the knowledge section
6. Save the project

Every conversation you have inside this project will automatically know the full spec. You should do all development work inside this project from now on.

Development Sessions

Each session below is a complete unit of work. Start a new conversation in your Project for each session. The conversations will get long and context-heavy; starting fresh per module keeps things clean and prevents confusion.

At the top of each conversation, you can start with:

> "We're building the SoulCore Framework prototype. The v3.0 spec is in the project knowledge. Today's session is: [session goal]."

Session 1 — Foundation Files

Goal: Create the files that every other module depends on.

Your prompt:

> "Create the following files for the SoulCore Framework prototype. For each file, write complete, production-quality Python code with full docstrings. Do not use any external libraries — stdlib only.
>
> File 1: `scf/__init__.py` — empty init file with a version string `__version__ = '0.1.0'`
>
> File 2: `scf/tier_interface.py` — the complete `Proposal` dataclass and `TierInterface` abstract class from Section 2.1 of the spec. Add a docstring to the `emotional_adjustments` field explicitly stating it is a delta value, not an absolute.
>
> File 3: `scf/storage.py` — the `StorageBackend` abstract base class from Section 3. For each abstract method, add a detailed docstring specifying exactly what the return type contains — every key and its meaning. This solves the return structure ambiguity noted in the spec review.
>
> After writing each file, show me the exact filename and full content."

What you do: For each file Claude produces, create a new file in your `soulcore` folder with that exact name and paste the content in. Save each file.

Verify: In the terminal, run `python -c "from scf import tier_interface; print('OK')"`. You should see `OK`.

Session 2 — Storage Backends

Goal: Implement both concrete storage backends.

Your prompt:

> "Implement both concrete storage backends for the SoulCore Framework.
>
> File 1: `scf/storage_sqlite.py` — `SQLiteBackend` implementing `StorageBackend` using Python's stdlib `sqlite3`. Use the schema from Section 3.1. Tags should be stored as JSON arrays using stdlib `json`. Include the corruption recovery logic described in Section 3.1. All methods must exactly match the interface defined in `scf/storage.py`.
>
> File 2: `scf/storage_btree.py` — `BTreeBackend` implementing `StorageBackend` using Python's stdlib `shelve` module as the fallback (not MicroPython btree — we're targeting Python 3.8+ for now and will add MicroPython support later). All data must be serialized with `json`. The interface must be identical to `SQLiteBackend`.
>
> File 3: `scf-devtools/tests/unit/test_storage_sqlite.py` — complete unit tests for SQLiteBackend covering: insert and retrieve emotional state, insert and retrieve memories (including tag handling), insert and retrieve learned directives, and the corruption recovery path.
>
> File 4: `scf-devtools/tests/unit/test_storage_btree.py` — the same test cases adapted for BTreeBackend.
>
> Use a tmp_path pytest fixture for all file-based tests so nothing persists between test runs."

What you do: Create all four files, paste the content, save.

Verify: In the terminal, run `pytest scf-devtools/tests/unit/test_storage_sqlite.py -v`. Every test should say PASSED. Then run `pytest scf-devtools/tests/unit/test_storage_btree.py -v`. Same result. If any test fails, copy the full error output and paste it back to Claude in the same conversation with: "This test failed. Here is the error. Please fix it."

Session 3 — The Condition DSL

Goal: The safe, eval-free DSL parser. This is the hardest module.

Your prompt:

> "Implement `scf/condition_dsl.py` — the complete condition DSL parser and evaluator for the SoulCore Framework. This is the most critical security component: it

> must NEVER use `eval()` or `exec()` under any circumstances.
>
> Implement:
>
> - `DSLSyntaxError(ValueError)` exception class
> - `evaluate_condition(condition: str, scope: dict) -> bool` using a recursive descent parser following exactly the grammar in Section 5.1
> - `evaluate_action(action: str, scope: dict) -> dict` for assignment statements
>
> The scope will contain: state variable names mapped to int values, `user_input` as a str, and `event_{id}` booleans.
>
> The `contains` operator must work as: `'phrase' in user_input_value.lower()`
>
> Any unrecognized token must raise `DSLSyntaxError` immediately — the parser must never silently fail or fall back to Python evaluation.
>
> Then implement `scf-devtools/tests/unit/test_condition_dsl.py` with the exact test cases specified in Section 15.2 of the spec, plus at least 5 additional edge cases you identify from the grammar.
>
> Show me the implementation first. I will review it before you show me the tests."

Note for you: After Claude shows you the implementation, respond: "This looks good. Now show me the test file." This two-step approach catches cases where Claude writes tests that pass only because they match a buggy implementation.

What you do: Create both files. Run `pytest scf-devtools/tests/unit/test_condition_dsl.py -v`. All tests should pass. Pay special attention to the injection test (`test_injection_raises`) — this one is non-negotiable.

Session 4 — Persona Loader

Goal: Parse and validate `.snp` files.

Your prompt:

> "Implement `scf/persona_loader.py` for the SoulCore Framework.
>
> The loader must:
>
> - Parse TOML format using Python's stdlib `tomllib` (Python 3.11+) with a fallback to the `tomli` third-party package for older versions. If neither is available, implement a minimal TOML subset parser for our specific schema (the spec

> guarantees no exotic TOML features).
>
> - Validate all rules from Section 7.1
> - Apply migration rules from Section 12 when the version is older than 1.1, using the semver tuple comparison from Section 7.2
> - Return the default neutral persona (defined as a constant in the file) on any failure, logging a WARNING
> - Compile all regex patterns at load time and store the compiled versions
>
> Then implement the following fixture files:
>
> `scf-devtools/tests/fixtures/pretorius.snp` — the full Dr. Septimus Pretorius persona from Section 7 of the spec
>
> `scf-devtools/tests/fixtures/neutral.snp` — a minimal valid persona with only required fields
>
> `scf-devtools/tests/fixtures/malformed_missing_identity.snp` — a TOML file missing the [identity] section
>
> `scf-devtools/tests/fixtures/v1.0_legacy.snp` — a v1.0 format persona that needs migration
>
> Then implement `scf-devtools/tests/unit/test_persona_loader.py` with the test cases from Section 15.2."

What you do: Create all files. Run `pytest scf-devtools/tests/unit/test_persona_loader.py -v`.

Checkpoint: After this session, you have: storage, DSL, and persona loading. These are the three foundations everything else rests on. Take a moment to make sure all tests pass before continuing.

Run: `pytest scf-devtools/tests/unit/ -v`

All tests in all files should pass. If anything is red, fix it before Session 5.

Session 5 — Memory Manager

Goal: The retrieval and decay logic.

Your prompt:

> "Implement `scf/memory.py` — the `MemoryManager` class for the SoulCore Framework.

> The MemoryManager accepts a `StorageBackend` instance at init and wraps it with the domain logic from Section 8.
>
> Implement:
>
> - `retrieve_memory(user_input: str, state: dict, rng: random.Random) -> Optional[dict]` following the three-step algorithm in Section 8.1 exactly
> - `apply_decay(variables_config: dict, state: dict) -> dict` implementing the decay formula from Section 8.2
> - Tokenization using this hardcoded stopword list: {'the','a','an','is','are','was','were', 'be','been','being','have','has','had','do','does','did','will','would','could','should', 'may','might','i','you','he','she','it','we','they','what','which','who','that','this','these', 'those','and','or','but','in','on','at','to','for','of','with','about'}
> - All four public methods from StorageBackend exposed as pass-throughs for convenience
>
> The emotional tag boost logic should reference state variable VALUES (integers), not the variable definitions. Variable `var` has boost applied if `state['variables'][var] > 70` and `var` appears in `memory['tags']`.
>
> Include unit tests in `scf-devtools/tests/unit/test_memory.py` covering: retrieval with exact event match, keyword scoring, emotional tag boost, decay calculation at t=0 and t=10 hours."

What you do: Create both files. Run `pytest scf-devtools/tests/unit/test_memory.py -v`.

Session 6 — Response Modulator

Goal: The deterministic voice filter.

Your prompt:

> "Implement `scf/modulator.py` — the `ResponseModulator` class for the SoulCore Framework.
>
> Implement as a class with a single static method: `apply(response: str, persona: dict, state: dict, rng: random.Random) -> str`
>
> Apply the five transformations from Section 6 in exact order. Key rules:
>
> 1. Word substitutions use `re.sub` with word boundaries

2. Forbidden lexicon replacements use case-insensitive regex
3. Characteristic phrase insertion: detect sentence boundaries as positions immediately after '.', '!', or '?' followed by whitespace or end-of-string. On a single-sentence response, append at end.
4. Emotional tag: if `emotional_indicators` is true, collect all variables whose value >= their threshold. If multiple triggered, select the one with the highest value. Tie-break: lexicographically earlier variable name. Append ONE tag only.
5. Imperfection: if triggered, insert at the beginning of a randomly chosen sentence boundary (same logic as #3).

Also implement `find_sentence_boundaries(text: str) -> List[int]` as a module-level function.

Then implement `scf-devtools/tests/unit/test_modulator.py` with the exact test cases from Section 15.2. The tests should set `imperfection_rate=0.0` and `characteristic_phrase_rate=0.0` for all tests except the ones specifically testing those features, and use `random.Random(0)` for all tests."

What you do: Create both files. Run `pytest scf-devtools/tests/unit/test_modulator.py -v`.

Session 7 — Session Handoff

Goal: The portable session snapshot.

Your prompt:

> "Implement `scf/session.py` — the `SessionSnapshot` dataclass and supporting functions for the SoulCore Framework.
>
> Implement:
>
> - `SessionSnapshot` dataclass from Section 9 (remove `recent_history_max` from the dataclass — this belongs in the runtime config, not the snapshot)
> - `_compute_checksum(snap: SessionSnapshot) -> str` using exactly this method: `hashlib.sha256(json.dumps({'variables': snap.variables, 'last_updated': snap.last_updated, 'entity_name': snap.entity_name}, sort_keys=True).encode()).hexdigest()`
> - `SessionImportError(Exception)` exception class

> The export and import methods will live on `Tier0StateMachine` — this file only defines the data structure and checksum logic.
>
> Then implement `scf-devtools/tests/unit/test_session_handoff.py` (the parts that test snapshot creation, JSON serialization, and checksum logic — the full roundtrip tests will be added after Tier0StateMachine is implemented)."

What you do: Create both files. Run `pytest scf-devtools/tests/unit/test_session_handoff.py -v`.

Session 8 — Tier 0 State Machine (Part A: Core)

Goal: The heart of the system. Split into two sessions because it's large.

Your prompt:

> "Implement the core of `scf/tier0_executor.py` — the `Tier0StateMachine` class for the SoulCore Framework. This is the most important file in the project.
>
> Session A covers: initialization, state management, event detection, transitions, and directive selection. Do NOT yet implement the Voice Gate, proposal application, or session export/import — those come next session.
>
> Implement:
>
> - `Tier0Config` dataclass from Section 4.1
> - `Tier0StateMachine.__init__(self, persona: dict, config: Tier0Config, storage_backend: Union[StorageBackend, str])` — when `storage_backend='memory'`, use a simple in-memory dict backend for testing
> - `_initialize_state(self) -> None` — reads variable definitions from persona, sets defaults
> - `_apply_decay(self) -> None` — applies decay formula using MemoryManager
> - `_detect_events(self, user_input: str) -> List[str]` — returns matched event ids
> - `_evaluate_transitions(self, user_input: str) -> None` — uses ConditionEvaluator from condition_dsl.py
> - `_select_directive(self, user_input: str) -> Optional[dict]` — implements the priority/tie-break logic from Section 4.3 and 4.4 steps 3

- `generate_response(self, user_input: str, **kwargs) -> str` — orchestrates steps 1-8 from Section 4.4, calling the modulator at step 6. The `kwargs` are available as template variables (e.g., `user_name`).

Use `random.Random(config.rng_seed)` if seed is set, else `random.Random()`.

Then implement `scf-devtools/tests/unit/test_tier0_state_machine.py` with the test cases from Section 15.2 that don't require proposal handling."

What you do: Create the file and test file. Run `pytest scf-devtools/tests/unit/test_tier0_state_machine.py -v`. When all tests pass, this is the biggest milestone in the project. The core works.

Session 9 — Tier 0 State Machine (Part B: Voice Gate + Proposals)

Your prompt:

"Continue implementing `scf/tier0_executor.py`. The core is already implemented. Now add:

- `_apply_voice_gate(self, base_response: str, proposal: Proposal) -> str` — the four-step gate from Section 2.2 in exact order: confidence check, forbidden_response_patterns check, axiom_guard_patterns check, accept. Log WARNING on any rejection with the reason.
- `generate_response_with_proposal(self, user_input: str, proposal: Proposal, **kwargs) -> str` — runs the full generation pipeline, then applies the Voice Gate on the amplifier's response_text (if any), runs the result through the modulator, returns final string. Distillation (apply_proposal) runs regardless of Voice Gate outcome.
- `apply_proposal(self, proposal: Proposal) -> None` — distillation logic from Section 10: validates and applies new_directives (with namespaced priority), new_knowledge, emotional_adjustments
- `export_session(self) -> SessionSnapshot` — from Section 9
- `import_session(self, snap: SessionSnapshot) -> None` — from Section 9, including checksum and entity validation

Then add the integration tests from `scf-devtools/tests/integration/test_amplifier_voice_gate.py` (exact test cases from Section 15.2) and complete the roundtrip tests in `test_session_handoff.py`."

What you do: Update the executor file (add the new methods without replacing existing ones — tell Claude to show you only the new methods to add). Create the integration test file. Run full test suite: `pytest scf-devtools/tests/ -v`.

Session 10 — Conversation Simulator

Goal: The YAML-driven test runner.

Your prompt:

> "Implement `scf-devtools/simulator.py` — the conversation simulator for the SoulCore Framework.
>
> The simulator:
>
> - Reads a YAML file (use stdlib `tomllib`? No — YAML is not stdlib. Use a simple line-by-line YAML parser for our specific schema, or ask me: should we add PyYAML as the only allowed dependency for devtools? Devtools are not part of the runtime, so external deps are acceptable there.)
>
> Wait — before implementing, answer this question in your response: should we use PyYAML for the simulator (it's a devtool, not runtime) or implement a minimal YAML parser? Recommend one and explain briefly.
>
> Then implement based on your recommendation. The simulator must:
>
> - Accept `--script path/to/script.yaml` and `--persona path/to/persona.snp` CLI arguments
> - Set `rng_seed` from the script's top-level field if present
> - Feed each `user` input to `Tier0StateMachine.generate_response()`
> - Assert `expect_state` (partial dict — only check listed variables) using exact comparison
> - Assert `expect_response_pattern` using `re.search(pattern, response)`
> - Print PASS/FAIL per turn with the actual response and state shown on failure
> - Exit with code 0 if all pass, 1 if any fail
>
> Also create the two simulator script files from Section 14 and 15.3."

What you do: Let Claude make the PyYAML decision (it will recommend PyYAML — correct call for a devtool). Install it: `pip install pyyaml`. Create the simulator and script

files. Run: `python scf-devtools/simulator.py --script scf-devtools/simulator_scripts/pretorius_basic.yaml --persona personas/pretorius.snp .`

Session 11 — Full Integration Test

Goal: Make sure everything works together.

Your prompt:

> "Write `scf-devtools/tests/integration/test_full_conversation.py` — a full integration test that:
>
> 1. Loads the pretorius.snp fixture
> 2. Creates a Tier0StateMachine with SQLiteBackend (temp file) and rng_seed=42
> 3. Runs a 10-turn conversation covering: a greeting, a moralizing statement, a second moralizing statement that pushes irritation above threshold, an insult, a thank-you, and 5 more varied inputs
> 4. After each turn, asserts that: (a) a non-empty string was returned, (b) the fallback was NOT used (unless you explicitly test a no-match case), (c) state variables are within their defined min/max
> 5. Exports a SessionSnapshot, creates a NEW Tier0StateMachine, imports the snapshot, sends one more message, and asserts the state is consistent with what was imported
> 6. Runs the entire test with no imports outside stdlib + pytest + the scf package
>
> This test should catch any integration bugs between the storage layer, state machine, modulator, and session system."

What you do: Create the file. Run `pytest scf-devtools/tests/integration/ -v` . If something fails here that passed in unit tests, paste the error back to Claude — this is a real integration bug worth fixing.

Session 12 — Sanity Check & Cleanup

Your prompt:

> "Run a final review pass. Here is the current state of the project:
>
> [Paste the output of: `find soulcore -name '*.py' -not -path '*/node_modules/*'`]

> [Paste the output of: `pytest scf-devtools/tests/ -v --tb=short`]
>
> Review:
>
> 1. Are there any `eval()` or `exec()` calls anywhere in scf/*.py? If so, flag them.
> 2. Does every module in scf/ have a module-level docstring?
> 3. Are there any bare `except:` clauses that swallow all exceptions without logging?
> 4. Does `generate_response` always return a string, even on exception?
> 5. Is the STOPWORDS constant defined in exactly one place (memory.py)?
>
> List any issues found and fix them. Then write a brief `README.md` in the project root that tells a developer: what the project is (one paragraph), how to run tests, how to run the simulator with the pretorius example."

After the Prototype — What Comes Next

When all 12 sessions are complete and all tests pass, you have a working Tier 0 soul engine. The following are natural next phases, in the order I'd recommend tackling them:

1. **Plain text adapter (`importers/plain_text.py`) — lets you import real chat logs**
2. **LocalLLMAmplifier stub** — a mock Tier 1 amplifier that returns canned Proposals, to test the Voice Gate end-to-end with real proposal flow
3. **PlatformDetector** — auto-selects SQLiteBackend vs BTreeBackend
4. **Benchmark script** — verifies the timing numbers from the spec
5. **Actual LocalLLM integration** — hook up a local model via `llama.cpp` or `ollama`
6. **FrontierAmplifier** — Anthropic API integration

The spec's critique issues that we deferred (full public API docs, formal stopword spec, etc.) are worth addressing in a v3.1 spec document once the prototype validates the architecture. Prototypes reveal which parts of the design actually needed more precision.

Quick Reference for the Human

Running all tests:

```
pytest scf-devtools/tests/ -v
```

Running just unit tests:

```
pytest scf-devtools/tests/unit/ -v
```

Running the simulator:

```
python scf-devtools/simulator.py --script scf-devtools/simulator_scripts/pretoriu
```

If a test fails: Copy the full output from the terminal, paste it into Claude in the relevant Project conversation, and say: "This test failed. Here is the complete error output. Please fix the issue and show me the corrected file."

If Claude's response is cut off: Say: "Please continue from where you left off."

If you're not sure what to do next: Paste your current terminal output and ask: "What's the next step?"