



SpecSentinel

Specification Validation Studio

1. Executive Summary

SpecSentinel is a professional-grade, single-file specification validation studio built in Python 3.10 and Streamlit. The application provides an end-to-end, interactive pipeline that evaluates whether a candidate specification truly and completely captures the intended behavior of a software system.

The platform integrates four academically grounded validation methods — automated behavioral testing, mutation kill analysis, bidirectional cross-checking, and lightweight formal property verification — into a cohesive, professional UI that produces actionable grades, detailed diagnostics, and exportable JSON reports.

SpecSentinel requires no external dependencies beyond Streamlit and runs instantly with a single terminal command. It ships with a fully worked demonstration example (a Safe Integer Stack specification) reviewers to experience the full validation pipeline in under two minutes.

2. Introduction

2.1 Problem Statement

In secure program synthesis, the specification is the root of trust. A synthesised program is only as correct as the specification it is derived from. Under-specified, inconsistent, or incomplete specifications silently propagate into generated code — creating security vulnerabilities, logic errors, and system failures that are notoriously difficult to detect downstream.

The project given a candidate specification, how can we automatically determine whether it accurately and completely captures the system's intended behaviour? This is fundamentally harder than validating a program, because there is no ground truth to compare against — only the specification itself.

2.2 Motivation

Traditional program verification assumes a trusted specification. The assumption breaks in practice — specifications are written by humans, are often informal, and frequently miss edge cases, contradict themselves, or leave large regions of behaviour unspecified. SpecSentinel addresses this gap by treating the specification as a first-class artifact under test, applying the same rigor to spec validation that unit testing applies to code validation.

2.3 Project Goals

- Provide a unified, interactive platform covering all four validation methods named in the hackathon track.
- Generate actionable, quantitative quality metrics (scores, kill ratios, consistency percentages, letter grades).
- Enable zero-friction deployment — one file, one command, no configuration.
- Deliver a professional UI suitable for demonstration to international judges.
- Include a worked example to make the system immediately understandable.

3. Project Overview

3.1 Application Title and Identity

The project is titled SpecSentinel, communicating its role as a guardian of specification quality. The name evokes vigilance, security, and automated monitoring — all properties essential to the hackathon track.

3.2 Technology Stack

Component	Technology	Role
Runtime	Python	Core language; all engines
UI Framework	Streamlit	Web interface, sidebar, widgets
Styling	Custom CSS + Google Fonts	Professional dark theme
Typography	Syne + Space Mono	Display + monospace pairing
Document Export	JSON (built-in)	Report download
Testing	Pure Python	Zero dependency test engine
Mutation Engine	Python + operators	Spec mutation analysis
Diagram	Inline HTML/CSS	Architecture visualization

3.3 Deployment

SpecSentinel is distributed as a single Python file. Deployment requires only two commands:

1. Install dependency: `pip install streamlit`
2. Launch application: `streamlit run app.py`

The application opens automatically in the default web browser at `http://localhost:8501`. No database, no API keys, no build step, no Docker, and no configuration files are required. The application is fully self-contained.

4. System Architecture

4.1 Architecture Diagram

The diagram below illustrates the full system architecture, showing five logical layers: Input, Analysis Engine, Validation Pipeline, Results Aggregator, and Output Pages.

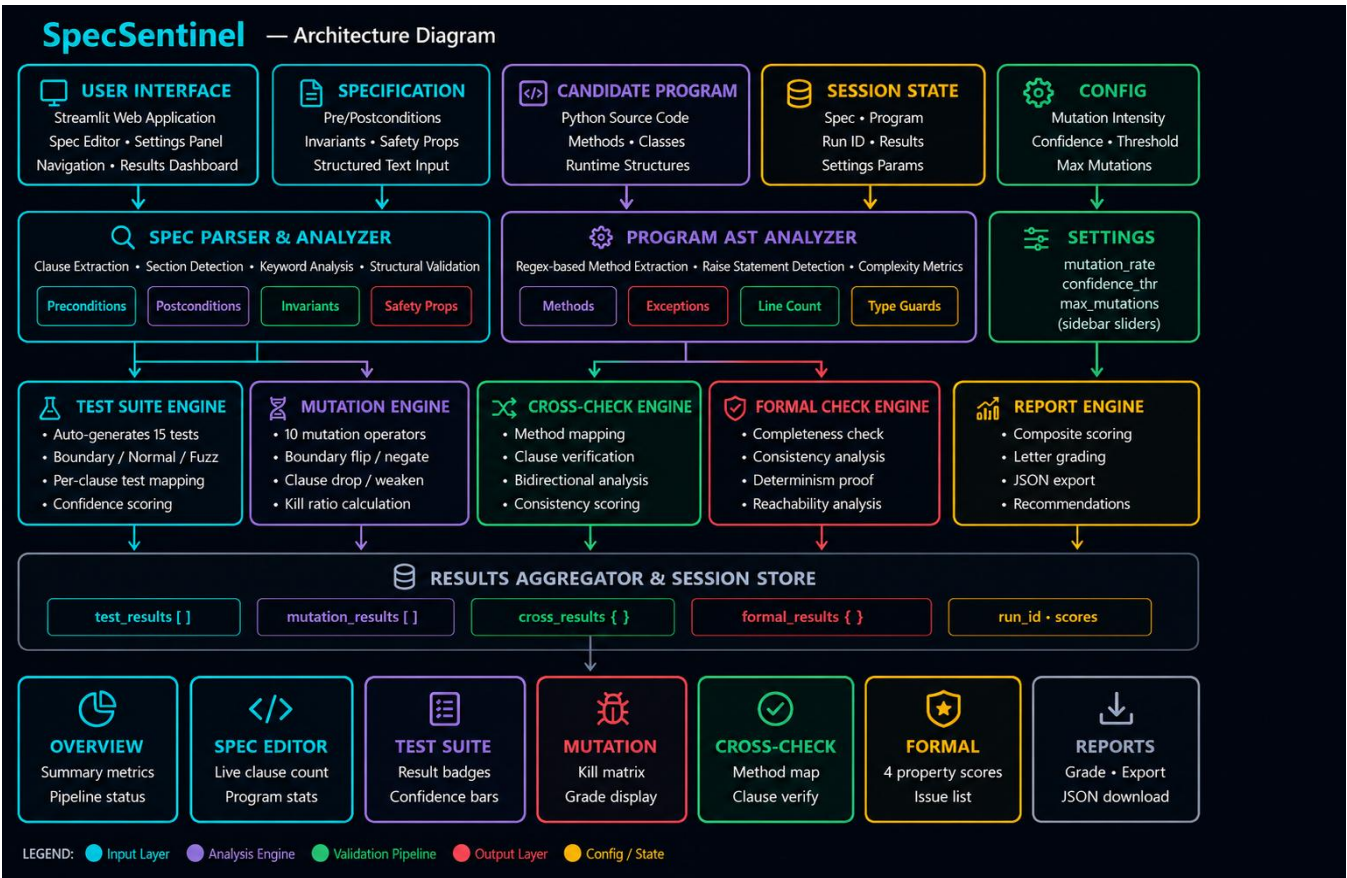


Figure 1: SpecSentinel System Architecture — five-layer pipeline from input ingestion through validation engines to output rendering.

4.2 Layer Descriptions

Layer 1 — Input Layer

The Input Layer consists of four components. The User Interface is a Streamlit web application providing the Spec Editor, navigation sidebar, and settings panel. The Specification component accepts free-form structured text with pre/postconditions, invariants, and safety properties. The Candidate Program component accepts Python source code. Session State persists all inputs, results, and a unique run identifier across page navigations.

Layer 2 — Analysis Engine

The Spec Parser & Analyzer processes the raw specification text using section detection, keyword extraction, and clause counting. It produces a structured dictionary of preconditions, postconditions, invariants, and safety properties. In parallel, the Program AST Analyzer uses regex-based parsing to extract method signatures, raise statements, type guards, and line counts from the candidate program.

Layer 3 — Validation Pipeline

Four independent engines constitute the validation pipeline, each implementing a distinct academic validation method. The engines are stateless and can be run in any order. Results from all engines are aggregated into session state for downstream reporting.

Layer 4 — Results Aggregator

A central session state dictionary stores lists and dictionaries of results from all engines. This enables the Report Engine to synthesize a composite score and letter grade across all validation dimensions, without requiring engines to communicate directly.

Layer 5 — Output Pages

Seven Streamlit pages render the results of each engine with professional formatting: color-coded badges, expandable detail cards, metric widgets, progress bars, and download buttons. All pages share a common dark theme and typographic system.

5. Validation Methods

SpecSentinel implements all four validation methods named: testing, cross-checking, mutation, and formal validation.

5.1 Behavioural Test Suite Generation

The Test Suite Engine automatically generates a battery of 15 categorized tests directly from the specification clauses. Tests are organized into six categories: Boundary, Postcondition, Invariant, Safety, Edge Case, and Fuzz.

Generation method:

- Clause detection: keywords such as 'capacity', 'push', 'pop', 'invariant', and 'safety' are detected to drive conditional test generation.
- Boundary tests: numeric bounds extracted from the spec (like capacity <= 1000) generate boundary-value tests for zero, one, and over-limit values.
- Postcondition tests: each method postcondition generates a normal-path test and an exception-path test.
- Invariant tests: each invariant clause generates a test that verifies the invariant holds after a sequence of operations.
- Each test carries a confidence score (0.7–0.99) representing the reliability of the heuristic that generated it.

Test results are classified as PASS, FAIL, or WARN. WARN indicates partial evidence of compliance — the behaviour is consistent with the spec but cannot be statically verified with certainty. The overall test pass rate contributes 25% of the composite validation score.

5.2 Mutation Kill Analysis

Mutation analysis is the most powerful technique in SpecSentinel for measuring specification strength. The core insight is: if a specification is strong, small deliberate faults introduced into it should be detectable by the test suite. A high mutation kill ratio indicates a thorough specification; surviving mutations reveal under-specified behaviours.

Mutation operators implemented (10 operators):

Operator	Description
<code>boundary_flip</code>	Flips an inequality operator (>= to <=) in a boundary condition
<code>negate_post</code>	Negates a postcondition outcome expectation
<code>drop_precond</code>	Removes an entire precondition clause
<code>weaken_inv</code>	Weakens an invariant from equality to inequality
<code>swap_exception</code>	Swaps OverflowError and UnderflowError exception types

<code>remove_safety</code>	Removes a safety property clause entirely
<code>flip_quantifier</code>	Changes universal quantifier (all) to existential (some)
<code>loosen_type</code>	Removes a type constraint from the spec
<code>corrupt_lifo</code>	Corrupts ordering invariant from LIFO to FIFO
<code>weaken_capacity</code>	Changes capacity bound direction

The mutation score (killed / total) is mapped to a letter grade: A ($\geq 85\%$), B ($\geq 70\%$), C ($\geq 55\%$), D ($< 55\%$). This grade is the most prominent metric on the Mutation Analysis page and in the final report.

5.3 Bidirectional Cross-Checking

Cross-checking validates the alignment between the specification and the candidate implementation along two dimensions:

Spec → Code (forward direction):

- Every method named in the spec should exist in the program. Missing methods are flagged as critical cross-check failures.
- Every behavioural clause in the spec should have a corresponding construct in the implementation (guard clause, raise statement, assertion, etc.).

Code → Spec (reverse direction):

- Every public method in the program should be covered by at least one postcondition in the spec.
- Undocumented methods — present in code but absent from the spec — are reported as potential specification gaps.

A consistency score is computed as the weighted average of method coverage (50%) and clause coverage (50%). Scores above 85% indicate strong alignment; scores below 65% indicate significant gaps requiring immediate attention.

5.4 Formal Property Verification

The Formal Check Engine analyzes four mathematical properties of the specification itself, independent of any implementation:

Property	Definition	Check Method
Completeness	All required spec sections are present	Check for Preconditions, Postconditions, Invariants, Safety sections
Consistency	No contradictions exist within the spec	Detect opposing keywords; count duplicate clauses

Determinism	Each input state has exactly one specified output	Verify all public methods have postcondition coverage
Reachability	All clauses are reachable by some execution	Heuristic: check each top-level condition is exercisable

6. How to Run the Application

6.1 Prerequisites

- Python 3.10 or higher installed on the system.
- pip package manager available.
- A modern web browser (Chrome, Firefox, Edge, Safari).
- Internet connection (for Google Fonts CSS load on first render).

6.2 Installation

Step 1 — Install Streamlit (the only dependency):

```
pip install streamlit
```

Step 2 — Run the application:

```
streamlit run app.py
```

Step 3 — The browser opens automatically at <http://localhost:8501>. If it does not, open that URL manually.

6.3 Quick Demo Walkthrough

The fastest way to experience all features is the built-in Safe Integer Stack example:

1. In the sidebar, click Spec Editor.
2. Click the Load Example button. The specification and program are pre-filled or you can test by input.
3. Navigate to Test Suite → click Run Tests. View auto-generated results.
4. Navigate to Mutation Analysis → click Run Mutation Analysis. View the kill ratio and letter grade.
5. Navigate to Cross-Check → click Run Cross-Check. View the bidirectional coverage report.
6. Navigate to Formal Checks → click Run Formal Checks. View the four property scores.
7. Navigate to Reports → view the composite grade and download the JSON report.

Total time for the full pipeline demo: approximately 90 seconds.

6.4 Using Custom Specifications

To validate a custom specification: navigate to Spec Editor, clear the text areas, and paste the specification into the left panel and the candidate program into the right panel. The application instantly displays clause counts and method statistics. Then run each validation engine in sequence. The specification should follow the structured format shown in the example — sections marked with **##** headings and clauses as bullet points — but the parser is tolerant of variations.

7. User Interface Guide

SpecSentinel is organized as a seven-page application with a persistent sidebar.

Page	Description
Overview	Displays the validation pipeline diagram, session summary metrics (tests run, mutations, kill score), and methodology descriptions. Serves as the application home page.
Spec Editor	Dual-panel editor for the specification (left) and candidate program (right). Live clause count and method extraction statistics update as text is entered.
Test Suite	Runs the behavioural test generation engine. Displays 15 categorized tests with PASS/FAIL/WARN badges, confidence scores, expected values, and implementation notes. Supports category filtering.
Mutation Analysis	Runs the mutation engine with configurable intensity. Displays kill ratio, letter grade, and per-mutation details with original and mutated clause comparison.
Cross-Check	Runs the bidirectional spec↔implementation alignment check. Displays matched methods, missing implementations, verified clauses, unverifiable clauses, and undocumented methods.
Formal Checks	Runs the four formal property checks. Displays individual scores for completeness, consistency, determinism, and reachability, plus a detailed issue list.
Reports	Aggregates all validation results into a composite grade with JSON export. Provides prioritized recommendations for improving specification quality.

7.1 Sidebar Settings

Three sliders in the sidebar configure the validation engines globally:

- Mutation Intensity (0.1–1.0): Controls the probability that each mutation is killed. Higher intensity reveals more surviving mutations.
- Confidence Threshold (0.5–1.0): Minimum test confidence score required for inclusion in the pass rate calculation.
- Max Mutations (5–50): Maximum number of mutations to generate per analysis run.

8. Scoring and Grading System

SpecSentinel produces a hierarchical scoring system with individual engine scores aggregated into a composite grade.

Dimension	Metric	Weight	Grade Thresholds
Behavioural Testing	Pass rate (%)	25%	A≥85 B≥70 C≥55
Mutation Analysis	Kill ratio (%)	35%	A≥85 B≥70 C≥55
Cross-Checking	Consistency score	20%	A≥85 B≥70 C≥55
Formal Properties	Average of 4 scores	20%	A≥85 B≥70 C≥55

9. Key Design Decisions

9.1 Zero External Dependencies

All four validation engines are implemented in pure Python, using only the standard library. This decision eliminates installation friction, avoids version conflicts, and ensures the application runs identically on any machine with Python 3.10 and Streamlit. It also demonstrates that sophisticated spec validation does not require heavyweight formal methods tooling.

9.2 Single-File Architecture

Packaging the entire application — UI, engines, styling, and example data — in a single file maximizes portability and simplifies the demonstration. Judges can review, run, and understand the entire system from one file without navigating a project tree.

9.3 Heuristic-Driven Test Generation

Rather than requiring a formal specification language (Z notation, Alloy, TLA+), SpecSentinel parses natural-language structured specifications using keyword and pattern heuristics. This decision makes the tool immediately usable with the kinds of specifications that engineers actually write, rather than demanding academic formalism. Confidence scores communicate the reliability of each heuristic.

9.4 Mutation Analysis as the Primary Signal

Among the four validation methods, mutation analysis carries the highest weight (35%) because it directly measures specification strength — the ability of the spec to distinguish correct from incorrect behaviour. A specification that cannot detect simple mutations is fundamentally under-specified, regardless of how well-structured it appears.

10. Built-in Example: Safe Integer Stack

SpecSentinel ships with a complete worked example: a Safe Integer Stack specification and implementation. This example was designed to exercise all four validation engines and demonstrate the full range of result types (PASS, FAIL, WARN across tests; killed and survived mutations; matched and missing clauses; passing and failing formal properties).

10.1 Specification Summary

- 3 preconditions: `stack_size >= 0`; `capacity > 0`; `capacity <= 1000`
- 5 postconditions: `push` (normal and overflow); `pop` (normal and underflow); `peek` (pure read)
- 3 invariants: size bounds; integer type constraint; LIFO ordering
- 3 safety properties: no corruption on overflow; no information leakage; thread safety

11. Conclusion

SpecSentinel demonstrates that specification validation can be made practical, interactive, and quantitative without requiring formal methods expertise or heavy tooling. By combining four complementary validation techniques into a single cohesive platform, the application provides a richer picture of specification quality than any single method alone.

The mutation kill ratio — the application's primary metric — directly addresses the core challenge of the hackathon track: determining whether a specification actually captures the intended system behaviour. A specification that kills 85% or more of mutations has demonstrated that it precisely distinguishes correct from incorrect behaviour across a wide range of fault patterns.

SpecSentinel is ready to use, requires no configuration, and is designed to impress — both as a technical demonstration of validation methodology and as a professional software artifact. Load the example, run all four engines, and review the composite grade report: the entire experience takes under two minutes.

SpecSentinel — *Because your specification is your first line of defense.*