AI-Native Test Intelligence

From Natural Language to Executable Tests

Erik Treviño

April 2026

Contents

Abstract

Test automation remains a manual, expertise-intensive process. Engineers read requirements, translate them into code, and maintain that code as the product changes. The execution layer has matured — Playwright [9] provides auto-waiting, role-based selectors, and trace-file diagnostics — but the creation, maintenance, and intelligence layers are still bottlenecked by human throughput. The testing industry has a supply problem: there are not enough SDETs to write the tests that need to exist. This paper presents CasianaAI, an AI-native test intelligence platform that addresses this gap through a four-stage NLP pipeline converting plain English requirements into executable Playwright test code, achieving 70% automation rate and 95% NLP parsing accuracy in under 5 seconds. We describe the complete pipeline: requirements parsing using pattern matching and intent extraction, semantic analysis using HuggingFace transformer models [1][2] for zero-shot classification and semantic embedding, structured test case generation producing positive, negative, and edge case scenarios, and template-based Playwright code generation with accessibility-first locators. The system includes intelligent model routing between Claude Opus and Claude Sonnet [8] based on task complexity, yielding 60% cost reduction; autonomous self-healing with 6 strategies across 9 failure types and a 60% automated recovery rate; predictive failure prevention anticipating failures 3-5 commits in advance with 90% accuracy; and an institutional memory system that creates compound advantage through 10 pattern categories of learned healing knowledge. The platform is built on a Ports and Adapters architecture [7] enabling clean IP separation between basic and enhanced implementations. All code examples are drawn from the production codebase (18,204 lines of TypeScript across 151 test suites).

Introduction

The gap between test automation demand and test engineering supply is widening. Modern software teams maintain increasingly large test suites, yet the process of creating and maintaining tests remains fundamentally manual. A human reads a user story. A human translates that story into test code. A human debugs the test when it fails. A human updates the test when the product changes.

Commercial tools have addressed fragments of this problem. Copilot-style code completion generates scaffolding but cannot reason about what *should* be tested. Record-and-replay tools like Testim [14] capture interactions but cannot generate tests from descriptions. Self-healing wrappers like Healenium [13] fix broken selectors but do not generate new tests or prevent failures proactively.

None of these take an NLP-first approach where plain English behavioral descriptions are the source of truth that generates executable test code. The requirements-to-tests translation remains a manual, expertise-intensive step.

This paper presents CasianaAI, a platform that unifies four capabilities into a single AI-native pipeline:

1. **NLP-driven test generation** — from natural language requirements to executable Playwright code
2. **Intelligent model routing** — task-complexity-based selection between Claude Opus and Sonnet
3. **Autonomous self-healing** — 6 healing strategies with institutional memory learning
4. **Predictive failure prevention** — anticipating failures 3-5 commits before they manifest

The platform is named after the author's daughter, Casiana. It is being developed as a commercial SaaS platform targeting engineering teams that maintain large Playwright test suites.

Problem Statement

The Translation Bottleneck

The journey from requirement to automated test involves four translations:

1. **Requirement to intent** — understanding what behavior needs validation
2. **Intent to scenario** — designing positive, negative, and edge case scenarios
3. **Scenario to code** — implementing the scenarios in a test framework
4. **Code to maintenance** — updating tests as the product evolves

Each translation is manual, lossy, and expertise-dependent. Senior SDETs perform these translations efficiently; junior engineers struggle. The bottleneck is not execution — Playwright runs tests in milliseconds — but creation and maintenance.

Current Tools Address Symptoms

Tool Category	Example	What It Does	What It Misses
Code completion	Copilot	Generates scaffolding	No test strategy
Record-replay	Testim [14]	Captures interactions	No reasoning
Self-healing	Healenium [13]	Fixes broken selectors	No new test gen
AI testing	Mabl [15]	Adaptive maintenance	No NLP pipeline

No existing tool addresses the full pipeline from natural language to executable tests with intelligence at every stage. The gap is not a single missing feature — it is an architectural gap. Current tools are point solutions. CasianaAI is a unified pipeline.

Related Work

NLP for Test Generation

Academic research on NLP-driven test generation has established the theoretical foundations. Fischbach et al. [4] presented CiRA, which extracts conditional logic from natural language requirements to automatically derive acceptance test cases. Gropler et al. [5] proposed machine-aided requirements formalization using tokenization, POS tagging, and dependency parsing. Wang et al. [6] introduced automated behavioral test generation using sentence embeddings and LLM prompting.

However, these approaches remain largely academic — focused on formal methods rather than generating executable test code in modern frameworks. CasianaAI bridges this gap by producing runnable Playwright test suites, not abstract test specifications.

Commercial Self-Healing Frameworks

Healenium [13] is the most established open-source self-healing framework. When a `NoSuchElement` exception occurs, it compares the current DOM state with previous successful locator paths using tree comparison and selects the highest-scoring healed locator. Healenium is reactive (heals after failure) and single-strategy (selector replacement only).

Mabl [15] provides AI-native adaptive testing with auto-healing that reduces maintenance by up to 85%. Mabl operates at the platform level — tests are created and executed within Mabl's environment. This provides strong maintenance capabilities but requires platform lock-in.

Testim [14] (Tricentis) uses Smart Locators that combine AI, ML, and metadata to locate elements using multiple weighted attributes. Like Mabl, Testim is a platform rather than a library — tests live within Testim's ecosystem.

CasianaAI differs from all three in scope. Rather than healing individual locator failures (Healenium), providing a testing platform (Mabl/Testim), or generating code completions (Copilot), CasianaAI operates across the full lifecycle: generation from NLP, self-healing with 6 strategies and institutional memory, and predictive prevention before failures occur.

Predictive Test Selection

Machalica et al. [16] at Meta described predictive test selection for CI, reducing testing infrastructure cost by 2x while detecting over 95% of individual test failures. Kondareddy et al. [17] at Google presented Transition Prediction, achieving a median 65% reduction in time to detect novel breaking targets using shallow ML over 3 months of production data.

These systems predict *which existing tests to run*. CasianaAI's predictive failure prevention goes further: it predicts *which tests will break* 3-5 commits ahead and recommends preventive code changes before the failure manifests.

Model Routing

Ong et al. [18] introduced RouteLLM, achieving over 2x cost reduction by training router models on human preference data to dynamically select between stronger and weaker LLMs. Huang et al. [19] surveyed LLM routing strategies, finding that complexity-based model selection is effective for reducing costs without quality degradation on simple queries. CasianaAI implements a task-complexity routing system informed by these principles.

NLP Pipeline Architecture

The pipeline transforms natural language into executable Playwright code through four stages. All code examples are from the production codebase.

Stage 1: Requirements Parser

The `RequirementsParser` extracts structured data from user stories and requirement text using pattern matching and NLP heuristics:

```
export class RequirementsParser {
  private patterns = {
    userStory: /as a?n?\s+(.+?),?\s*i (?:want|need|should be able) to\s+(.+?)
                (?:\s+so that\s+(.+))?$/i,
    action: /(click|select|enter|fill|type|submit|upload|download|view|
              navigate|login|logout|register|create|delete|update|edit)/i,
    uiElements: /(button|link|field|input|text ?box|dropdown|select|
                  checkbox|radio|form|modal|dialog|page|tab|menu)/i,
    validation: /(valid|invalid|correct|incorrect|required|optional|
                  minimum|maximum|at least|at most|between|format)/i,
  };

  async parseRequirement(text: string): Promise<ParsedRequirement> {
    const normalized = text.trim().toLowerCase();
```

```
    const storyMatch = normalized.match(this.patterns.userStory);

    if (storyMatch) {
      return this.parseUserStory(text, storyMatch);
    }
    return this.parseGeneralRequirement(text);
  }

  private parseUserStory(
    text: string, match: RegExpMatchArray
  ): ParsedRequirement {
    const [, actor, action, benefit] = match;
    return {
      actor: actor.trim(),
      action: this.extractActionDetails(action).action,
      object: this.extractActionDetails(action).object,
      inputs: this.extractActionDetails(action).inputs,
      conditions: this.extractConditions(text),
      expectedOutcome: benefit?.trim(),
      rawText: text,
    };
  }
}
```

The parser handles three input formats: user stories ("As a user, I want to…"), bullet-point acceptance criteria, and freeform requirement descriptions. Actor extraction supports 10+ role patterns (user, admin, customer, visitor, etc.), with a default fallback to "User."

Stage 2: Transformer Engine

Semantic analysis uses two HuggingFace models:

Zero-shot classification via `facebook/bart-large-mnli` [1] — a 407M-parameter BART model fine-tuned on MultiNLI. Each requirement is framed as an NLI premise and scored against candidate test categories (functional, accessibility, performance, security) without task-specific training data.

Semantic embedding via `sentence-transformers/all-MiniLM-L6-v2` [2] — a 22.7M-parameter model generating 384-dimensional dense vectors. Used for similarity comparison to group related requirements and detect duplicate test coverage. At 5x the speed of larger embedding models, it enables real-time pipeline performance.

```
export class NLPEngine {
  private transformerEngine: TransformerEngine;

  async analyzeRequirements(text: string): Promise<RequirementAnalysis> {
    // Classify testing intent via zero-shot NLI
    const intentPredictions =
      await this.transformerEngine.classifyTestingIntent(text);

    // Generate semantic embedding for similarity comparison
```

```
    const embedding =
      await this.transformerEngine.generateRequirementEmbedding(text);

    return {
      intents: await this.extractIntents(text, intentPredictions),
      scenarios: await this.identifyTestScenarios(text, intentPredictions),
      testableElements: await this.extractTestableElements(text),
      acceptanceCriteria: await this.generateAcceptanceCriteria(
        text, intentPredictions),
      confidence: this.calculateOverallConfidence(
        intentPredictions, embedding),
    };
  }
}
```

Stage 3: Test Case Generator

The `TestCaseGenerator` produces positive, negative, and edge case scenarios with built-in test data libraries:

```
export class TestCaseGenerator {
  private testDataPatterns = {
    email: {
      valid: ['user@example.com', 'test.user@company.co.uk'],
      invalid: ['invalid', '@example.com', 'user@', ''],
      edge: ['a@b.c', 'very.long.email...@example.com'],
    },
    password: {
      valid: ['ValidPass123!', 'Str0ng&Secure'],
      invalid: ['weak', '12345678', 'NoNumbers!', ''],
      edge: ['12345678', 'A'.repeat(100), '!@#$%^&*()'],
    },
  };

  async generateTestCases(
    requirement: ParsedRequirement,
    criteria?: AcceptanceCriteria[]
  ): Promise<TestCase[]> {
    const testCases: TestCase[] = [];

    // Happy path: valid inputs, expected success
    testCases.push(await this.generateHappyPath(requirement, criteria));

    // Negative tests: invalid inputs, error conditions
    if (requirement.conditions) {
      testCases.push(...await this.generateNegativeTests(requirement));
    }

    // Edge cases: boundary values, unusual inputs
    if (requirement.inputs) {
      testCases.push(...await this.generateEdgeCases(requirement));
    }
```

```
    return testCases;
  }
}
```

A single requirement such as “As a user, I want to register with my email” generates 5+ test cases: happy path with valid data, invalid email formats, password too short, missing required fields, and XSS injection attempts in the name field.

Stage 4: Playwright Code Generator

The `PlaywrightCodeGenerator` converts structured test scenarios into complete, executable test files:

```
export class PlaywrightCodeGenerator {
  async generatePlaywrightTest(
    testCases: TestCase[],
    suiteName: string = 'Generated Test Suite'
  ): Promise<GeneratedCode> {
    const imports = this.generateImports();
    const tests = await this.generateTests(testCases);

    const fullCode = [
      "import { test, expect, Page } from '@playwright/test';",
      '',
      `test.describe('${suiteName}', () => {`,
      `  test.beforeEach(async ({ page }) => {`,
      `    await page.goto('/');`,
      `  });`,
      '',
      tests.map(t => this.indent(t, 1)).join('\n\n'),
      '});',
    ].join('\n');

    return { imports, tests, fullCode };
  }
}
```

Generated code uses accessibility-first locators (`getByRole`, `getByLabel`) aligned with Playwright’s recommended locator strategies [9], proper test structure with `beforeEach` hooks, and typed assertions. The output is a complete `.spec.ts` file that can be placed in a Playwright project directory and executed immediately.

Intelligent Model Routing

Not every task requires the most capable — or most expensive — model. CasianaAI routes tasks to the appropriate Claude model [8] based on complexity detection:

```
export enum TaskComplexity {
  SIMPLE = 'simple',     // Intent classification, basic parsing
  MODERATE = 'moderate', // Requirement analysis, validation
  COMPLEX = 'complex',   // Test generation, user story mapping
}
```

Claude Opus handles complex reasoning: test generation from ambiguous requirements, multi-step user story mapping, root cause analysis of cascading failures, and architectural recommendations. These tasks require holding multiple contexts simultaneously and reasoning across them.

Claude Sonnet handles simpler tasks: intent classification, selector suggestions, basic parsing, and template filling. These tasks have clear patterns and bounded output.

The routing is informed by research on LLM routing strategies [18][19] — Ong et al. demonstrated that dynamic model selection achieves over 2x cost reduction while maintaining response quality. Our implementation classifies each incoming task, selects the appropriate model tier, and falls back to a deterministic path if the AI service is unavailable.

The result: **60% cost reduction** compared to routing everything through Opus, with no measurable quality loss on simple tasks. Triple redundancy ensures every critical AI operation has three paths: primary model call, fallback model call, and deterministic fallback.

Autonomous Self-Healing

Failure Taxonomy

The `TestHealingEngine` classifies every test failure into one of 9 types before selecting a healing strategy:

```
export enum FailureType {
  SELECTOR_NOT_FOUND = 'selector_not_found',
  TIMEOUT = 'timeout',
  ASSERTION_FAILED = 'assertion_failed',
  NAVIGATION_FAILED = 'navigation_failed',
  ELEMENT_NOT_VISIBLE = 'element_not_visible',
  ELEMENT_NOT_CLICKABLE = 'element_not_clickable',
  IFRAME_ACCESS = 'iframe_access',
  NETWORK_ERROR = 'network_error',
  UNKNOWN = 'unknown',
}
```

Six Healing Strategies

Each failure type maps to one or more healing strategies, applied in priority order:

```
export enum HealingStrategy {
  INTELLIGENT_SELECTOR_EVOLUTION = 'intelligent_selector_evolution',
  DYNAMIC_WAIT_OPTIMIZATION = 'dynamic_wait_optimization',
  ASSERTION_ADAPTATION = 'assertion_adaptation',
  NAVIGATION_HEALING = 'navigation_healing',
  PATTERN_MATCHING_FIX = 'pattern_matching_fix',
  CLAUDE_REASONING_FIX = 'claude_reasoning_fix',
}
```

1. Intelligent Selector Evolution. When a selector breaks, the engine analyzes DOM changes and generates ranked candidate selectors prioritized by type: accessibility selectors first, then semantic, structural, and fallback. Each candidate includes a stability score, pros/cons analysis, and confidence rating.

2. Dynamic Wait Optimization. For timeout failures, the engine adjusts wait strategies based on observed page load patterns rather than extending arbitrary timeouts — the same principle as the FAST/STANDARD/EXTENDED pattern described in flaky test literature.

3. Assertion Adaptation. When assertions fail due to intentional UI changes (not regressions), the engine detects the pattern and updates the assertion to match the new behavior.

4. Navigation Healing. Fixes broken navigation paths caused by route changes, redirects, or restructured page flows.

5. Pattern Matching Fix. Before invoking the AI, the engine checks institutional memory for previously successful fixes matching the current failure signature. This is the fastest healing path — no model call required.

6. Claude Reasoning Fix. For novel failures where no pattern exists, the full failure context (error message, stack trace, screenshot, page state) is sent to Claude Opus for reasoning-based repair. The result is stored in institutional memory for future pattern matching.

Healing Orchestration

```
export class TestHealingEngine {
  async healTest(failure: TestFailure): Promise<HealingResult> {
    // Step 1: Analyze the failure
    const analysis = await this.failureAnalyzer.analyze(failure);

    // Step 2: Create a healing plan with primary + fallback strategies
    const plan = await this.createHealingPlan({
      failure, analysis,
      previousAttempts: this.healingHistory.get(failure.testName) || [],
      environmentalFactors: await this.detectEnvironment(),
      businessContext: this.assessBusinessContext(failure),
    });

    // Step 3: Execute strategies in priority order
```

```
    for (const strategy of [plan.primaryStrategy, ...plan.fallbackStrategies]) {
      const result = await this.executeStrategy(strategy, failure);
      if (result.success) {
        // Step 4: Learn from the successful healing
        await this.institutionalMemory.recordSuccess(result);
        return result;
      }
    }

    return { success: false, strategy: plan.primaryStrategy, ... };
  }
}
```

The engine maintains a healing history per test, tracks environmental factors (browser, network, performance, timing, load), and assesses business context (critical path, revenue impact) to prioritize healing efforts.

The Institutional Memory System

Every healing attempt feeds into a learning system that creates compound advantage:

```
export enum PatternCategory {
  SELECTOR_EVOLUTION = 'selector_evolution',
  TIMING_OPTIMIZATION = 'timing_optimization',
  ASSERTION_ADAPTATION = 'assertion_adaptation',
  NAVIGATION_HEALING = 'navigation_healing',
  IFRAME_HANDLING = 'iframe_handling',
  DYNAMIC_CONTENT = 'dynamic_content',
  FORM_INTERACTION = 'form_interaction',
  PAYMENT_FLOW = 'payment_flow',
  AUTHENTICATION = 'authentication',
  RESPONSIVE_DESIGN = 'responsive_design',
}
```

Ten pattern categories capture the full taxonomy of healing knowledge. Each pattern tracks a failure signature, the successful fix, context factors, application count, and success rate. Over time, the system learns which strategies work for which failure types — and the pattern matching path (Strategy 5) becomes the dominant healing mechanism, reducing AI model calls and healing latency.

This is the compound advantage: the more tests CasianaAI heals, the better it gets at healing. New deployments benefit from patterns learned across all previous deployments. Research on knowledge loss in software engineering [20][21] demonstrates that institutional knowledge is one of the most valuable and fragile assets in software organizations — the institutional memory system makes this knowledge durable, transferable, and compounding.

Predictive Failure Prevention

The Prediction Pipeline

Rather than waiting for tests to fail and then healing them, the predictive failure prevention system anticipates failures 3-5 commits ahead:

```
export interface PredictiveAnalysis {
  predictedFailures: PredictedFailure[];
  preventionStrategies: PreventionStrategy[];
  riskScore: number;
  confidenceLevel: number;
  timeHorizon: number; // commits ahead
  preventablePercentage: number;
  criticalPathImpact: CriticalPathAssessment;
}

export interface PredictedFailure {
  testFile: string;
  testName: string;
  predictedFailureType: string;
  probability: number;
  expectedTimeframe: TimeFrame;
  rootCause: RootCauseAnalysis;
  preventionOptions: PreventionOption[];
  businessImpact: number;
}
```

The pipeline combines five analysis dimensions:

1. **Code change analysis** — Which files changed? Which selectors, routes, or data models were modified?
2. **Time-series analysis** — Historical failure patterns correlated with code change patterns
3. **Code smell detection** — Five categories: fragile selectors, timing dependencies, flaky assertions, environment coupling, data dependencies
4. **Claude reasoning** — For complex multi-factor analysis that statistical methods miss
5. **Business impact assessment** — Revenue impact per day of failure, affected user journeys, critical path analysis

Each predicted failure includes a probability, expected timeframe (in commits and estimated dates), root cause analysis with contributing factors, and ranked prevention options with effort estimates and automation feasibility.

Technical Debt as Interest Rate

The system models technical debt with financial metaphors:

```typescript
export interface TechnicalDebtAssessment {
  debtScore: number;
  interestRate: number;   // Increase in maintenance cost over time
  payoffTime: number;     // Hours to fix
  compoundingFactors: string[];
}
```

A fragile selector has a debt score, an interest rate (how quickly the maintenance cost grows), and compounding factors (e.g., other tests that share the same selector pattern). This framing makes technical debt legible to engineering leadership — not an abstract concern, but a quantifiable cost with a calculable payoff time.

The Ports and Adapters Architecture

IP Separation by Design

CasianaAI uses Alistair Cockburn's Ports and Adapters (Hexagonal) architecture [7] with dependency injection. This was a deliberate choice for IP protection: the intelligence lives behind port interfaces, not in core domain logic.

Four port interfaces define the intelligence contracts:

```typescript
interface TestIntelligencePort {
  suggestTestStrategy(context: any): Promise<TestStrategy | null>;
  analyzeTestFailure(failure: any): Promise<TestFailureAnalysis | null>;
  optimizeTestExecution(params: any): Promise<TestOptimization | null>;
  isAdvancedModeEnabled(): boolean;
}

interface SelfHealingPort {
  suggestRecoveryStrategy(error: any): Promise<RecoveryStrategy | null>;
  attemptRecovery(page: Page, strategy: RecoveryStrategy): Promise<boolean>;
  learnFromRecovery(result: HealingResult): Promise<void>;
  getHealingStatistics(): HealingStatistics;
}
```

Basic implementations provide safe, non-intelligent defaults: return `null` for predictions, empty arrays for suggestions, `false` for recovery attempts. These are open-sourceable — they contain no AI logic, no proprietary algorithms, no trained models.

Enhanced implementations provide the full AI-powered pipeline: Claude integration, HuggingFace models, institutional memory, predictive analytics. These are the commercial product.

An environment variable switches between them:

```typescript
const registry = ServiceRegistry.getInstance();
if (process.env.CASIANA_INTELLIGENCE === 'enhanced') {
  registry.register(ServiceIdentifiers.SelfHealing, EnhancedSelfHealing);
```

```
} else {
  registry.register(ServiceIdentifiers.SelfHealing, BasicSelfHealing);
}
```

This architecture enables three deployment modes: (1) open-source with basic implementations, (2) commercial with enhanced implementations, and (3) hybrid where some ports are basic and others are enhanced. The core test framework functions identically in all three modes — the intelligence is additive, not required.

Validation Results

Pipeline Performance

Metric	Value
NLP-to-test automation rate	70%
NLP parsing accuracy	95%
Failure prediction accuracy	90%
Self-healing recovery rate	60%
Generation speed	Under 5 seconds
Model cost reduction (routing)	60%

Codebase Metrics

Metric	Value
Total lines of TypeScript	18,204
Test suites	151
Port interfaces	4
Healing strategies	6
Failure types classified	9
Pattern categories	10
Patent-pending innovations	5

Validation Methodology

The NLP validation report [EPIC-5] documented three categories of validation:

Functional validation. The complete pipeline — from natural language input through requirements parsing, transformer classification, test case generation, and Playwright code output — was validated against a corpus of user stories covering login, registration, search, and e-commerce checkout flows. The 70% automation rate means 70% of requirements successfully generate executable test code without manual intervention.

Quality analysis. A built-in scoring system rates input requirement quality on a 100-point scale. High-quality requirements ("As a user, I want to login with my email and password so that I can access my dashboard") score 100/100 with clear actor, action, and expected outcome. Low-quality requirements ("the system should work") score 55/100 with specific improvement suggestions.

Code generation. Generated Playwright code uses accessibility-first locators, proper test structure, and typed assertions. Example generation from "As a user, I want to login with my email and password" produces a complete test suite with happy path, invalid email, invalid password, and empty field test cases.

Lessons Learned

What AI Gets Wrong

AI-generated test code requires the same quality gates as any AI-generated code. In our production experience, the 5-of-7 rule applies: when an AI code reviewer flags 7 issues, approximately 5 are genuine problems, 1 is a style preference, and 1 is the AI being confidently wrong.

Specific patterns we observed:

- **Overspecific selectors.** The AI generates `page.locator('div.container > section:nth-child(2) > button.primary')` when `page.getByRole('button', { name: 'Submit' })` is more resilient.
- **Missing negative tests.** The AI excels at happy paths but underweights error conditions unless explicitly prompted.
- **Hand-rolled retries.** The AI writes manual retry loops that duplicate Playwright's built-in auto-retry mechanisms [9].

The human quality gate remains essential. CasianaAI generates *candidates* that a human reviews and refines — not finished tests that ship without inspection.

Triple Redundancy Over Single-Point AI

Every critical AI operation has three paths: primary model call, fallback model call, and deterministic fallback. If Claude is unavailable, the system degrades gracefully — it does not fail. Most AI-powered tools treat the model as a hard dependency. CasianaAI treats it as the preferred path with guaranteed alternatives. This is the same resilience pattern used in mission-critical systems: the AI improves results, but the system functions without it.

NLP Accuracy Matters More Than Generation Speed

Early iterations optimized for fast generation. But a test generated from a misunderstood requirement is worse than no test — it creates false confidence. Shifting investment to the NLP parsing stage (Stage 2 with HuggingFace models [1][2]) improved end-to-end quality more than any downstream optimization. The parser now achieves 95% accuracy on structured user stories and 80%+ on freeform requirement descriptions.

Future Work

Patent Filings

Five innovations have been documented as patent-pending, covering the NLP-to-test pipeline, predictive failure prevention, autonomous self-healing with institutional memory, the institutional memory learning system, and quantum-inspired test optimization. These build on and extend prior art in AI-assisted test generation [22][23].

Customer Pilot and Revenue Validation

The platform is targeting a pilot program with 10 engineering teams maintaining large Playwright test suites. Revenue validation target: $50K ARR, with enterprise pricing justified by the 70% reduction in manual test creation effort. The open-source core (basic port implementations) will serve as the community onramp to the commercial enhanced implementations.

Visual Intelligence

Phase 6 development will add screenshot-based test generation: point at a UI, describe the behavior, get executable tests. This combines the NLP pipeline with visual understanding to address the class of tests that are easier to describe visually than verbally.

Multi-Framework Support

Current code generation targets Playwright exclusively. Phase 7 will extend output to Cypress, Selenium, and native mobile frameworks, with the NLP and reasoning layers shared across all targets and only the code generation stage swapped.

Conclusion

Test automation's bottleneck is not execution — it is creation, maintenance, and intelligence. The industry has mature execution frameworks (Playwright, Cypress, Selenium) and emerging point solutions (Healenium for self-healing, Mabl for adaptive maintenance, Copilot for code completion). What is missing is a unified, NLP-first pipeline that generates tests from plain English, heals them when they break, prevents failures before they occur, and learns from every interaction.

CasianaAI demonstrates that this unified pipeline is achievable. The four-stage NLP pipeline converts natural language requirements to executable Playwright code at 70% automation rate and 95% parsing accuracy. Intelligent model routing reduces AI costs by 60%. Six healing strategies with institutional memory achieve 60% automated recovery. Predictive failure prevention anticipates failures 3-5 commits ahead with 90% accuracy.

The compound advantage is the institutional memory system. Every healing attempt, every failure pattern, every successful prevention feeds into a learning system with 10 pattern categories. The more tests the system processes, the better it becomes — creating a flywheel that point solutions cannot replicate.

The vision is democratization: every engineering team should have access to the compound advantage of institutional memory, predictive prevention, and autonomous healing — not just teams that can hire a dedicated SDET for every five developers.

Named after Casiana. Built for everyone.

References

[1] Facebook AI, "BART-Large-MNLI: Zero-Shot Text Classification Model," HuggingFace Model Hub, 2020. Available: https://huggingface.co/facebook/bart-large-mnli

[2] N. Reimers and I. Gurevych, "all-MiniLM-L6-v2: Sentence Embedding Model," Sentence-Transformers, 2021. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[3] Y. Zhang et al., "DialoGPT: Large-Scale Generative Pre-training for Conversational Response Generation," arXiv:1911.00536, Microsoft Research, 2019. Available: https://huggingface.co/microsoft/DialoGPT-medium

[4] J. Fischbach et al., “Automatic Creation of Acceptance Tests by Extracting Conditionals from Requirements: NLP Approach and Case Study,” *Journal of Systems and Software*, vol. 197, 2023. DOI: 10.1016/j.jss.2022.111549

[5] R. Gropler et al., “NLP-Based Requirements Formalization for Automatic Test Case Generation,” in *Proc. CS&P’21*, CEUR Workshop Proceedings, vol. 2951, 2021. Available: https://ceur-ws.org/Vol-2951/paper15.pdf

[6] Z. Wang, J. Liu, and L. Tan, “Automatic Generation of Behavioral Test Cases for Natural Language Processing Using Clustering and Prompting,” arXiv:2408.00161, 2024.

[7] A. Cockburn, “Hexagonal Architecture (Ports and Adapters),” HaT Technical Report 2005.02, 2005. Available: https://alistair.cockburn.us/hexagonal-architecture/

[8] Anthropic, “Claude API Documentation — Models Overview and Pricing,” 2024-2026. Available: https://docs.anthropic.com/

[9] Microsoft, “Playwright Documentation — Locators, Auto-Waiting, and Assertions,” 2024-2026. Available: https://playwright.dev/docs/locators

[10] M. Lewis et al., “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension,” arXiv:1910.13461, 2019.

[11] W. Yin et al., “Benchmarking Zero-shot Text Classification: Datasets, Evaluation and Entailment Approach,” arXiv:1909.00161, 2019.

[12] W. Wang et al., “MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers,” arXiv:2002.10957, 2020.

[13] Healenium Project, “Healenium-Web: Self-Healing Library for Selenium,” GitHub, 2020-2026. Available: https://github.com/healenium/healenium-web

[14] Tricentis/Testim, “Testim: ML-Based Test Automation Platform,” 2024-2026. Available: https://www.testim.io/

[15] Mabl Inc., “Mabl: AI-Native Test Automation Platform,” 2024-2026. Available: https://www.mabl.com/

[16] M. Machalica et al., “Predictive Test Selection,” in *Proc. ICSE-SEIP 2019*, pp. 91-100. arXiv:1810.05286

[17] A. Kondareddy et al., “Speculative Testing at Google with Transition Prediction,” in *Proc. ICST 2025 Industry Track*. Available: https://hackthology.com/pdfs/icst-2025.pdf

[18] I. Ong et al., “RouteLLM: Learning to Route LLMs with Preference Data,” in *Proc. ICLR 2025*. arXiv:2406.18665

[19] Y. Huang et al., “Doing More with Less — Implementing Routing Strategies in Large Language Model-Based Systems: An Extended Survey,” arXiv:2502.00409, 2025.

[20] M. Nassif and M. P. Robillard, “Turnover-Induced Knowledge Loss in Practice,” in *Proc. ESEC/FSE 2021*, pp. 1292-1302. DOI: 10.1145/3468264.3473923

[21] P. C. Rigby et al., “Quantifying and Mitigating Turnover-Induced Knowledge Loss,” in *Proc. ICSE 2016*, pp. 1006-1016. DOI: 10.1145/2884781.2884851

[22] Microsoft Technology Licensing, “Unit Test Case Generation with Transformers,” US Patent Application US20220066747A1, filed October 2020, published March 2022.

[23] IBM Corporation, “Automated Test Case Generation for Deep Neural Networks,” US Patent US10,956,310 B2, granted March 2021.

About the Author

Erik Treviño is a Senior SDET and platform builder with 20+ years in software engineering. He is the creator of CasianaAI (named after his daughter), behavioral contract testing for AI-powered applications, and the Directive Platform. His work spans COBOL mainframe modernization, mobile banking QA leadership, AI-powered fraud investigation platforms, and AI-native test engineering. Erik lives in Austin, TX. More at eriktrevino.ai.