Entity Model & Relationships

1.1 Core Entities

Organization

- **Definition**: Top-level tenant entity representing a company or team
- **Attributes**:
  - Organization ID (unique identifier)
  - Name, subscription plan, license counts
  - Settings (product preferences, security policies)
  - User management and roles
- **Relationships**:
  - Contains multiple Users
  - Contains multiple Projects
  - Has organization-level Secrets, Variables, and Settings

User

- **Definition**: Individual who interacts with the platform
- **States**: Active, Inactive, Suspended, Invited
- **Roles**: Admin, User, Guest
- **Attributes**:
  - User ID, email, name
  - Access tokens, authentication methods
- **Relationships**:
  - Belongs to Organization
  - Can access multiple Projects
  - Creates/Authors/Edits Test Cases, Test Runs
  - Has user-level Secrets

Project

- **Definition**: Container for organizing test assets and activities
- **Attributes**:
  - Project ID, name, description
  - Creation date, last modified
  - Project settings and configurations
- **Relationships**:
  - Belongs to Organization

    - Contains Test Cases, Test Runs, Milestones, Datasets, Reports, Insights
    - Linked to external systems (Jira, ADO projects)

Folder

- **Definition**: Organizational unit within a project for grouping either test cases or test runs (two separate folder types)
- **Types**:
    - **Test Case Folders**: Used to organize and group test cases
    - **Test Run Folders**: Used to organize and group test runs
- **Attributes**:
    - Folder ID, name, path
    - Parent folder reference
    - Order/position in hierarchy
    - Type (test case folder or test run folder)
- **Relationships**:
    - Belongs to Project
    - Has parent Folder of same type (except root)
    - Contains child Folders of same type (nested up to N levels)
    - Test Case Folders contain Test Cases
    - Test Run Folders contain Test Runs

Test Case

- **Definition**: A discrete testing scenario with steps and expected outcomes
- **Types**:
    - **KaneAI Authored**: AI-generated test case with automation script generated
    - **Manual Test Steps**: Traditional test case with manual test steps (typical test management tool usage)
    - **BDD Scenarios**: Test case with BDD (Behavior-Driven Development) scenarios (typical test management tool usage)
- **States**: (Can be custom as well defined in organisation settings)
    - **Unverified**: Not yet authored by KaneAI
    - **Ready**: Complete and ready for execution
    - **Faulty**: Issues in authoring in KaneAI and hence code cannot be generated
    - **Live**: Reviewed and approved
    - **Archived**: Historical reference
- **Attributes**:
    - Test Case ID (unique, immutable)
    - Title, description, pre-conditions
    - Type (Functional, Regression, Smoke, etc.)
    - Priority (Highest, High, Medium, Low, Lowest)
    - Status (see States above)
    - Version number
    - Tags

    - Custom fields
    - Test Steps with Expected Outcomes
- **Relationships**:
    - Belongs to Project and Folder
    - Has multiple Test Steps
    - Has multiple Versions (version history)
    - Linked to Issues (Jira/ADO)
    - Part of Test Runs
    - Can reference Modules
    - Uses Variables, Parameters, Secrets
    - Can be generated by Test Scenario (via Generate entity)
    - Has Code (automation scripts) if KaneAI Authored type and successfully authored
- **Version Management**:
    - Each save creates a new version
    - **AI-Generated Commit Message**: When a test case is saved, an LLM prompt generates a commit message summarizing the changes
    - Commit message stored with each version for change tracking
    - Version history shows all changes with commit messages
    - Users can view and compare versions
    - Rollback to previous versions supported

Test Step

- **Definition**: Individual action or assertion within a test case
- **Types**: (Only relevant for KaneAI authored test case types)
    - **Action Step**: Performs an action (click, type, navigate, scroll)
    - **Assertion Step**: Validates a condition
    - **API Step**: Makes API calls
    - **Database Step**: Executes queries
    - **Module Step**: References a reusable module with multiple steps inside it.
    - **JavaScript Step**: Executes custom JS
- **Attributes**:
    - Step number/order
    - Instruction (natural language)
    - Step type
    - Expected outcome
    - Failure condition (Fail immediately, Fail but continue, Warn but continue)
    - Timeout settings
- **Relationships**:
    - Belongs to Test Case
    - May reference Variables - Local, Environment, Global
    - May reference Secrets
    - May reference Parameters
    - May reference Modules

    - Has execution results in Test Instance

Module

- **Definition**: Reusable collection of test steps that can be shared across both KaneAI-authored and Manual Test Steps type test cases
- **Creation Methods**:
    - **Direct from Modules Page** (`kaneai.lambdatest.com/modules`):
        - Simple modules with string-based test steps and expected outcomes
        - Cannot contain API, variables, parameters, DB, or JS steps
        - Compatible with both Manual and KaneAI test cases
    - **From KaneAI Agent** (pause agent and select consecutive steps):
        - Can include already-authored or queued steps
        - Supports advanced step types (variables, parameters, secrets, API, DB, JS)
        - Allows editing and adding steps (creates new version)
- **Attributes**:
    - Module ID, name, description
    - Version number
    - Tags
    - Test Steps with Expected Outcomes per step
    - Linked Projects (can be associated with multiple projects)
    - Creation method (direct or KaneAI-authored)
- **Relationships**:
    - Contains Test Steps
    - Has Version history
    - Referenced by Test Cases (both Manual and KaneAI types)
    - Cannot reference other Modules (nesting not allowed)
    - Uses Variables, Secrets, Parameters, API, JS, DB (only for KaneAI-authored modules)
- **Notes**:
    - Expected outcomes are irrelevant for KaneAI test cases (only used in Manual Test Steps)
    - Can be project-level or organization-level
    - Version updates when test steps, expected outcomes, or other fields are modified

Session (KaneAI Session)

- **Definition**: Interactive authoring environment for creating and developing test cases in KaneAI
- **States**:
    - **Queued**: Waiting for resources to become available
    - **Running**: Active authoring session in progress

  - **Completed**: Session finished successfully
  - **Interrupted**: Session stopped or terminated before completion
- **Attributes**:
  - Session ID
  - User ID (who started the session)
  - Platform type (Web/Mobile)
  - Start time, end time
  - Duration
  - Session capabilities and settings
  - Concurrency limit (based on subscription)
- **Relationships**:
  - Belongs to User
  - Belongs to Organization (subscription limits)
  - Starts an Agent
  - Creates/Edits Test Cases (via Agent)

Agent

- **Definition**: AI-powered execution engine within a session that performs test authoring and editing operations
- **Trigger Modes**:
  - **Authoring New Test**: Creating a new test case from scratch
  - **Editing Existing Test (Playground)**: Modifying an already created test case
- **Execution Modes**:
  - **Foreground Mode**: User has control over the session, user chooses when to save
  - **Background Mode**: Agent has control, auto-saves when authoring completes or fails/gets stuck
  - **Mode Transition**: User can take control in Background mode, transitioning to Foreground mode
- **Agent States** (State Management):
  - **Setting Up**: Initializing the agent environment and loading configuration
  - **Recording**: Recording steps via manual interaction (manual interaction enabled)
  - **Idle**: Ready to accept natural language instructions or slash commands
  - **Authoring**: Natural language instructions being analyzed, authored, and performed
  - **Error**: Error during step authoring or replaying, agent pauses for user action
  - **Pausing**: Transition state when user clicks Pause button
  - **Draft**: Paused state for editing instructions, adding steps in between, or creating modules
  - **Replaying**: Replaying already authored steps
- **Instruction Input Modes**:
  - **Structured Mode**: Deterministic test steps added explicitly
  - **Adaptive Mode**: AI explores and figures out steps autonomously

    - **Slash command**: For custom actions including API, JS, Network assertions, DB query, creating a variable, secret or parameter, totp authentication, image injection, video injection or uploading files.
- **Attributes**:
    - Agent ID
    - Session ID (parent session)
    - Current state
    - Execution mode (Foreground/Background)
    - Trigger mode (New/Edit)
    - Test case being authored/edited
    - Instructions/steps in progress
- **Instruction-Level Operations** (in Draft state):
    - Edit instruction
    - Delete instruction
    - Run from here (execute from specific instruction)
    - Run till here (execute up to specific instruction)
    - Update failure conditions
    - Update step timeout
- **Relationships**:
    - Belongs to Session
    - Authors/Edits Test Cases
    - Contains Test Steps/Instructions
    - References Modules
    - Uses Variables, Secrets, Parameters
    - Triggers Code Generation (on successful authoring)

Code (Automation Code)

- **Definition**: Generated automation script from a successfully authored KaneAI test case
- **States**:
    - **Generating**: Code generation in progress
    - **Validating**: Generated code being validated
    - **Success**: Code generated and validated successfully
    - **Failure**: Code generation or validation failed
- **Supported Frameworks/Languages**:
    - **Web**:
        - Selenium - Python (PyTest)
        - Selenium - Java (coming soon)
        - Playwright - JavaScript (coming soon)
        - Playwright - Python (coming soon)
        - Cypress - JavaScript (coming soon)
        - WebdriverIO - JavaScript (coming soon)
    - **Mobile**:
        - Appium - Python (PyTest)

        - Appium - Java (coming soon)
        - Appium - JavaScript (coming soon)
- **Attributes**:
    - Code ID
    - Test Case ID (parent)
    - Test Case Version
    - Framework/Language
    - Generated code content
    - Generation timestamp
    - Status (see States above)
    - Validation errors (if any)
- **Generation Triggers**:
    - **Automatic**: Triggered when agent completes authoring successfully
    - **Manual**: User can manually generate code for a test case (if authoring was successful)
- **Generation Conditions**:
    - All test steps must be successfully authored
    - No skipped steps allowed
    - No failed steps allowed
    - Test case must be KaneAI Authored type
- **Relationships**:
    - Belongs to Test Case
    - Generated after successful Agent authoring
    - Used by HyperExecute for test execution
    - Can have multiple Code instances (different frameworks/languages)

Test Run

- **Definition**: A collection of test cases and configurations for execution
- **States**:
    - **Not started**: Initial state
    - **In Progress**: Currently executing
    - **Failed**: Execution failed
    - **Passed**: Execution passed
- **Attributes**:
    - Test Run ID
    - Name, description
    - Configuration (browser, OS, device)
    - Concurrency settings
    - Environment selection
    - Schedule details
- **Relationships**:
    - Belongs to Project
    - Contains Test Instances
    - Triggers HyperExecute Jobs

    - Has execution results
    - Can have associated Schedules (for KaneAI Generated type)

Schedule

- **Definition**: Recurring execution configuration for KaneAI Generated test runs on HyperExecute
- **Scope**: Only available for test runs with KaneAI Authored test cases (type: KaneAI Generated)
- **States**:
    - **Active**: Schedule is enabled and will trigger at configured times
    - **Paused**: Schedule exists but won't trigger until resumed
    - **Disabled**: Schedule is inactive
- **Attributes**:
    - Schedule ID
    - Name
    - Cron expression (schedule timing)
    - Timezone
    - Test run reference
    - Execution configuration overrides (optional)
    - Last run timestamp
    - Next run timestamp
    - Created by (User)
    - Notification settings
- **Relationships**:
    - Belongs to Test Run (specifically KaneAI Generated type)
    - Belongs to Project (nested within Test Runs section)
    - Triggers HyperExecute Jobs on schedule
    - Creates Test Instances on each execution
- **Location in UI**: Schedules appear as a sub-entity within Test Runs in the project navigation
- **Notes**:
    - Non-KaneAI Generated test runs cannot have schedules (manual execution only)
    - Multiple schedules can be created for the same test run
    - Each schedule execution creates a new test run instance with results

Test Instance

- **Definition**: A specific execution of a test case + configuration within a test run
- **Attributes**:
    - Instance ID
    - Execution configuration
    - Start/end times
    - Duration

    - Result status
- **Relationships**:
    - Belongs to Test Run
    - References Test Case
    - Has Step Results
    - Generates artifacts (logs, screenshots, videos)

Variable

- **Definition**: Dynamic data placeholder
- **Types**: String, JSON, Smart Variables
- **Scopes**: Local (test), Global (organization), Environment-specific
- **Attributes**:
    - Variable name (unique within scope)
    - Value
    - Data type
    - Persist flag (local to global)
    - Environment association
- **Relationships**:
    - Belongs to Organization or User
    - Used by Test Cases and Modules
    - Can reference other Variables

Secret

- **Definition**: Encrypted sensitive data
- **Scopes**: User-level, Organization-level
- **Attributes**:
    - Secret key (identifier)
    - Encrypted value
    - Access control
    - Audit trail
- **Relationships**:
    - Belongs to User or Organization
    - Referenced in Test Cases via {{secrets.x}} syntax
    - Stored in HashiCorp Vault

Environment

- **Definition**: Configuration set for different deployment stages
- **Attributes**:
    - Environment ID, name
    - Variable mappings
    - Configuration overrides

- **Relationships**:
    - Contains environment-specific Variables
    - Referenced by Test Runs
    - Applied to Test Executions

Milestones

- **Definition**: Central organizational tool for grouping and tracking test runs, particularly valuable for monitoring feature launches
- **States**:
    - **Active**: Currently in progress
    - **Completed**: Finished and marked as complete
- **Attributes**:
    - Milestone ID
    - Name (required)
    - Description
    - Tags
    - Owner (assigned team member)
    - Duration (expected timeframe)
    - Attachments
    - Associated test runs
- **Relationships**:
    - Belongs to Project
    - Contains Test Runs
    - Has Owner (User)
    - Referenced by Reports

Dataset

- **Definition**: Collection of structured data used to manage and reuse input values for data-driven testing
- **Types**:
    - **Default Dataset**: Auto-generated during test authoring (immutable)
    - **Custom Dataset**: User-created or copied from default (editable)
- **Attributes**:
    - Dataset ID, name
    - Parameters (columns/placeholders)
    - Rows (test scenarios with different data values)
    - Data types (text, numbers, etc.)
    - Version history
    - Creation method (Manual, AI Autofill, CSV Import)
- **Relationships**:
    - Belongs to Organization
    - Used by Test Cases
    - References Parameters

    - Has Version history

Report

- **Definition**: Analytical tool providing actionable insights from testing activities, measuring test coverage and tracking progress
- **Types**:
    - **Execution Reports**: Analyze test execution results and identify patterns
    - **Traceability Reports**: Show alignment between test cases, requirements, and defects
- **Attributes**:
    - Report ID
    - Name (required)
    - Description
    - Report type
    - Filters (test runs, dates, test/issue/run metadata)
    - Recurring option (automated email delivery)
    - Recipients
    - Schedule frequency
- **Relationships**:
    - Belongs to Project
    - References Test Runs
    - References Test Cases
    - References Issues (Jira/ADO)
    - References Milestones

Generate (AI Test Case Generator)

- **Definition**: AI-powered service that generates test scenarios and test cases from various input sources
- **Input Sources**:
    - Text descriptions
    - Images/Screenshots
    - Audio/Voice inputs
    - Documents (PDFs, Word docs)
    - Jira issues
    - Azure DevOps work items
- **Attributes**:
    - Generate ID
    - Input type
    - Input content/reference
    - Generation timestamp
    - User who triggered generation
    - Processing status
- **Relationships**:

    - Triggered by User
    - Belongs to Project and Folder (where test cases will be created)
    - Generates Test Scenarios
    - Uses LLM Processing (Azure OpenAI)

Test Scenario

- **Definition**: Grouping of related test cases generated by AI, representing a testing objective or feature area
- **Categories**:
    - **Must-have**: Critical scenarios that must be tested
    - **Should-have**: Important scenarios recommended for testing
    - **Could-have**: Optional scenarios for comprehensive coverage
- **Test Types**:
    - **Functional Tests**: Generated for all scenarios except the last
    - **Non-Functional Tests**: Generated for the last scenario (performance, security, accessibility, etc.)
- **Attributes**:
    - Scenario ID
    - Title/Name
    - Category (Must-have/Should-have/Could-have)
    - Test type (Functional/Non-Functional)
    - Description
    - Number of test cases (2-10 per scenario)
- **Relationships**:
    - Belongs to Generate
    - Contains Test Cases (2-10)
    - Generated as group but test cases can be individually selected

1.2 Entity Relationship Diagram

- `Organization (1) ───────┬─────── (*) Users`
- `                        ├───── (*) Sessions (KaneAI authoring sessions)`
- `                        ├───── (*) Projects ──────┬────── (*) Folders ──────┬────── (*) Test Cases`
- `                        │                         │ └───── (*) Generate (AI generation requests)`
- `                        │                         ├───── (*) Test Runs ───── (*) Test Instances`

- │   ├─── (*) Milestones
- │   ├─── (*) Datasets
- │   └─── (*) Reports
- ├─── (*) Modules
- ├─── (*) Secrets
- ├─── (*) Variables/Environments
- └─── (*) Parameters
-
- User (1) ──────────── Triggers (*) Generate
-
- Generate (1) ───────┬─── Belongs to Project and Folder
- ├─── (*) Test Scenarios (2-10 scenarios)
- └─── Uses LLM Processing (Azure OpenAI)
-
- Test Scenario (1) ──┬──── (*) Test Cases (2-10 cases per scenario)
- └─── Has Category (Must-have/Should-have/Could-have)
-
- Session (1) ─────┬──── (1) User (creator)
- └─── (1) Agent
-
- Agent (1) ──────┬──── (*) Test Steps/Instructions (in progress)
- ├─── (1) Test Case (being authored/edited)
- ├─── (0..1) KaneAI Test Plan (for automation in Create & Automate)
- └─── (*) Modules (referenced)

-
- Test Case (1) ────┬──── (*) Test Steps
- ├──── (*) Code (only if KaneAI Authored and successfully authored)
- ├──── (*) Issues (Jira/ADO)
- ├──── (*) Runs
- ├──── (*) Versions
- └──── (0..1) KaneAI Test Plan (for AI-generated tests)
-
- Code (1) ──────────┬──── Belongs to Test Case (specific version)
- ├──── Has State (Generating/Validating/Success/Failure)
- ├──── Framework/Language selection
- └──── Multiple Code instances per test case (different frameworks)
-
- Test Run (1) ─────┬──── (*) Test Instances
- └──── (0..1) HyperExecute Job (only for KaneAI Generated)
-
- Module (1) ────────┬──── (*) Test Steps
- └──── (*) Versions

Entity State Transitions & Lifecycles

2.1 Session (KaneAI Session) Lifecycle

- `[Queued] → [Running] → [Completed]`
- `              ↓              ↓`
- `       [Test Case Created] [Test Case Saved]`
- `              ↓`
- `          [Interrupted]`
- `              ↓`
- `       [Test Case Deleted]`

State Transition Rules

- **Queued → Running**: Resources become available, session initialization complete
- **Running**: Test Case is automatically created when session starts
- **Running → Completed**: User finishes authoring and session ends successfully
- **Completed**: Test Case is saved to Test Manager
- **Running → Interrupted**: Session timeout, user cancellation, or system error
- **Interrupted**: Test Case is automatically deleted

Concurrency Rules

- Maximum concurrent sessions determined by organization subscription
- New sessions queue when concurrency limit reached
- Session timeout after idle period (typically 2-10 minutes based on tier)

2.2 Agent State Management Lifecycle

- `                               [Setting Up]`
- `                                     ↓`
- `                          ┌──────[Recording]──────┐`
- `                          │          ↓            │`
- `                          │   (Manual Interaction`
- `                          │    button/Toggle)     │`

```
•            │         ↓           │
•            └────→ [Idle] ←───────┘
•                   ↓   ↑
•            (Input step/    (Authoring
•             slash command)  complete)
•                   ↓   ↑
•                [Authoring]
•                   ↓   ↓
•            (Error/Abort)  (Exit & Run from here)
•                   ↓   ↓
•            ┌──── [Error] ────┐
•            │       ↓         │
•            │  (Click Replay  │
•            │   button)       │
•            │       ↓         │
•            │  [Replaying] ◄──┘
•            │       ↓
•            │  (Step fails)
•            │       ↓
•            └───────┘
•
• [Pause button clicked from any state]
•                    ↓
•                [Pausing]
•                    ↓
```

- `                    (Pausing complete)`
- `                           ↓`
- `                        [Draft]`
- `                           ↓`
- `           ┌───────────────┼───────────────┐`
- `           ↓               ↓               ↓`
- `      [Recording]     [Replaying]    [Resume → Idle]`

State Transition Rules

Initial States

- **Setting Up**: Agent initializes environment, loads configuration
- **Setting Up → Recording**: Setup complete, agent starts in Recording mode (manual interaction enabled)

Recording ↔ Idle

- **Recording → Idle**: Click Manual Interaction button (toggle off) or Stop Recording
- **Idle → Recording**: Click Manual Interaction button (toggle on)
- These transitions are bidirectional and can happen at any time

Idle State

- **Idle → Authoring**: User inputs natural language step or slash command (Structured or Adaptive mode)
- **Authoring → Idle**: Authoring complete, ready for next instruction
- Idle is the primary state for receiving natural language instructions

Authoring State

- **Authoring → Error**: Error occurs during step execution or user aborts step
- **Authoring → Error**: User clicks "Exit step & Run from here"
- **Authoring → Idle**: Step execution completes successfully

Error State

- **Error → Replaying**: User clicks Replay button or Replay from top
- Errors can occur during Authoring or Replaying
- Agent pauses in Error state waiting for user action

Replaying State

- **Replaying → Error**: Step fails during replay
- **Replaying → Replaying**: User clicks Replay button again
- **Replaying → Idle**: Resume session, replay complete

Pausing & Draft State

- **Any State → Pausing**: User clicks Pause button
- **Pausing → Draft**: Pausing process completes
- **Draft State** allows instruction-level operations:
  - Edit instruction
  - Delete instruction
  - Run from here
  - Run till here
  - Update failure conditions
  - Update step timeout
- **Draft → Recording**: Click Manual Interaction button
- **Draft → Replaying**: Click Replay button
- **Draft → Idle**: Resume session

Execution Mode Behaviors

Foreground Mode

- User controls all state transitions
- Manual save required (user chooses when to save)

Background Mode

- Agent controls state transitions
- Auto-saves when authoring completes successfully (returns to Idle)
- Auto-fails if Error state and agent gets stuck
- User can take control: Background → Foreground (transitions to Draft state via Pausing)

Trigger Mode Behaviors

- **Authoring New Test**: Starts with empty test case, begins from Start → Setting Up → Recording
- **Editing Existing Test (Playground)**: Loads existing test steps into Draft state

2.3 Test Case Lifecycle

- `[Unverified] → [Ready] → [Live] → [Archived]`
- `       ↓          ↓         ↓         ↓`
- `   [Faulty]  [Executed] [Modified] [Reference Only]`
- `       ↓          ↓         ↓`
- `   [Fixed]   [Results] [New Version]`

State Transition Rules

- **Unverified → Ready**: KaneAI authoring completes successfully, code generated
- **Unverified → Faulty**: Issues during KaneAI authoring, code cannot be generated
- **Faulty → Ready**: Issues fixed, successful code generation
- **Ready → Live**: Test case reviewed and approved for production use
- **Ready/Live → Modified**: Test case edited (creates new version)
- **Live → Archived**: Test case no longer in active use
- **Any State → New Version**: When edited (creates version history)

Note: States can be custom-defined in organization settings. The above are default states.

2.4 Test Run Lifecycle

- `[Not Started] → [In Progress] → [Passed]`
- `                      ↓            ↓`
- `                  [Failed]    [Results & Artifacts]`
- `                      ↓`
- `                  [Retry]`

State Transition Rules

- **Not Started → In Progress**: Execution initiated, HyperExecute job created
- **In Progress → Passed**: All test instances completed successfully
- **In Progress → Failed**: One or more test instances failed
- **Failed → Retry**: Manual re-execution or automatic retry triggered
- **Passed/Failed**: Results, logs, screenshots, and videos generated

Detailed Workflows & Interactions

3.1 Test Authoring Workflow (KaneAI)

Phase 1: Session & Agent Initialization

- `1. User initiates session`
- `   ├── Selects platform (Web/Mobile)`
- `   ├── Chooses trigger mode:`
- `   │   ├── Author New Test: Start from scratch`
- `   │   └── Edit Existing Test (Playground): Load existing test`
- `   ├── Selects execution mode:`
- `   │   ├── Foreground Mode: User controls save timing`
- `   │   └── Background Mode: Agent auto-saves on completion`
- `   ├── Configures capabilities (browser/device, network, geolocation)`
- `   └── Session queued (if concurrency limit reached)`
-
- `2. Session starts → Test Case created`
- `   ├── Session moves to Running state`
- `   ├── Agent initialized within session`

- └── Test Case automatically created (deleted if session interrupted)
-
- 3. Agent initialization
- ├── Agent State: Setting Up
- ├── Loads configuration and environment
- ├── Agent State: Recording (manual interaction enabled by default)
- └── Ready for user input

Phase 2: Test Step Authoring (Agent-Driven)

- Agent State Flow:
- [Recording] ↔ [Idle] → [Authoring] → [Draft]/[Error] → [Replaying]
-
- 1. Recording Mode (Manual Interaction)
- ├── User performs actions on browser/device
- ├── Actions captured and converted to steps
- ├── Toggle off: Recording → Idle
- └── Can return to Recording anytime from Idle or Draft
-
- 2. Idle Mode (Natural Language Input)
- ├── Agent ready for instructions
- ├── Input methods:
- │   ├── Structured Mode: Explicit deterministic steps
- │   ├── Adaptive Mode: AI explores and figures out steps

- │   └── Slash Commands: /api, /js, /db, /network, /totp, etc.
- ├── Toggle on: Idle → Recording
- └── Input received → Authoring
-
- 3. Authoring Mode
- ├── AI analyzes natural language instruction
- ├── Generates and executes test step
- ├── Step execution outcomes:
- │   ├── Success → Return to Idle (ready for next instruction)
- │   ├── Error/Abort → Error state
- │   └── Exit & Run from here → Error state
- └── Variables/Secrets/Parameters can be created/referenced
-
- 4. Error State
- ├── Agent pauses, displays error
- ├── User actions:
- │   ├── Click Replay button → Replaying
- │   ├── Click Pause button → Pausing → Draft
- │   └── Fix and retry → Error resolved → Idle
- └── Background mode: Auto-fails if stuck
-
- 5. Pausing & Draft State
- ├── User clicks Pause button from any state
- ├── Agent State: Pausing → Draft
- ├── Draft mode allows:

- │   ├── Edit instruction
- │   ├── Delete instruction
- │   ├── Run from here / Run till here
- │   ├── Update failure conditions
- │   ├── Update step timeout
- │   ├── Add steps in between
- │   └── Create/reference modules
- └── Exit Draft:
-     ├── Resume → Idle
-     ├── Manual Interaction button → Recording
-     └── Replay button → Replaying
-
- 6. Replaying Mode
- ├── Replays authored steps
- ├── Outcomes:
- │   ├── Success → Idle (ready to continue)
- │   ├── Step fails → Error state
- │   └── User pauses → Draft
- └── Can replay multiple times

Phase 3: Validation & Assertions

- During Authoring (in any mode):
- ├── Add assertions via slash commands or natural language:
- │   ├── /network: Network assertions
- │   ├── /api: API response validation

- │   ├── /db: Database query results
- │   └── UI element validations
- ├── Set failure conditions per step:
- │   ├── Fail immediately
- │   ├── Fail but continue
- │   └── Warn but continue
- └── Configure step timeouts

Phase 4: Save & Complete

- Foreground Mode:
- ├── User manually clicks Save at any time
- ├── Select Project and Folder
- ├── Add metadata (name, description, tags)
- ├── Set initial status (Unverified/Ready)
- ├── Test Case saved to Test Manager
- ├── Session → Completed
- └── Code Generation (if all steps successful):
-     ├── Code State: Generating
-     ├── Code State: Validating
-     ├── Code State: Success (automation script available)
-     └── Test Case Type: KaneAI Authored
-
- Background Mode:
- ├── Agent auto-saves when authoring completes
- ├── Returns to Idle after successful authoring

- ├── Auto-saves and marks Faulty if stuck in Error
- ├── Session → Completed (test case saved)
- └── Code Generation (if all steps successful):
-     ├── Automatic trigger on successful authoring
-     ├── Code State: Generating → Validating → Success
-     ├── Automation script available for execution
-     └── Test Case Type: KaneAI Authored
-
- Code Generation Failure:
- ├── If any step was skipped or failed: No code generated
- ├── Test Case Status: Faulty
- └── User can manually trigger code generation after fixing issues
-
- Session Interrupted:
- └── Test Case automatically deleted

3.2 Module Creation & Management Workflow

3.2.1 Direct Module Creation (from Modules Page)

- 1. Navigate to Modules page
-    └── URL: kaneai.lambdatest.com/modules
-
- 2. Create new module
-    ├── Click "Create Module" button
-    ├── Enter module details:

- │   ├── Name (required)
- │   ├── Description
- │   ├── Tags
- │   └── Link Projects (select multiple projects)
-
- 3. Add test steps
- ├── Add Step button
- ├── For each step:
- │   ├── Enter test step details (string format)
- │   └── Enter expected outcome (string format)
- ├── Reorder steps as needed
- └── Delete steps if needed
-
- 4. Save module
- ├── Module created with version v1
- ├── Type: Manual/Basic module
- ├── Limitations:
- │   ├── No API steps
- │   ├── No variables or parameters
- │   ├── No database queries
- │   ├── No JavaScript execution
- │   └── No secrets
- └── Compatible with both Manual Test Steps and KaneAI test cases
-
- 5. Link to projects

- `    └── Module available in linked projects`
- `        └── Can be referenced in test cases`

3.2.2 Module Creation from KaneAI Agent

- `1. During active KaneAI session`
- `   ├── Agent in any state (Idle, Authoring, Recording, Draft)`
- `   └── Test steps already authored or in queue`
-
- `2. Pause agent`
- `   ├── Click Pause button`
- `   ├── Agent State: Pausing → Draft`
- `   └── Test steps visible in session`
-
- `3. Select consecutive steps`
- `   ├── Select multiple adjacent steps (authored or queued)`
- `   ├── Right-click or use action menu`
- `   └── Option: "Convert to Module"`
-
- `4. Configure module`
- `   ├── Enter module details:`
- `   │   ├── Name (required)`
- `   │   ├── Description`
- `   │   ├── Tags`
- `   │   └── Link Projects (select multiple projects)`
- `   ├── Review included steps:`

- │   ├── Can include action steps
- │   ├── Can include assertion steps
- │   ├── Can include API steps
- │   ├── Can include database steps
- │   ├── Can include JavaScript steps
- │   ├── Can include variables and parameters
- │   ├── Can include secrets (e.g., {{secrets.org.password}})
- │   └── Cannot include other module steps (no nesting)
- └── Confirm creation
-
- 5. Module created
- ├── Version: v1
- ├── Type: KaneAI-authored module
- ├── Contains advanced step types
- ├── Test steps retain their execution logic:
- │   ├── Variables resolved at runtime
- │   ├── Secrets fetched securely
- │   ├── API calls executed
- │   ├── DB queries executed
- │   └── JS code executed
- ├── Expected outcomes captured (relevant for Manual Test Steps only)
- └── Available in linked projects
-
- 6. Resume agent
- ├── Selected steps replaced by Module step in test case

- ├── Agent State: Draft → Idle or Recording
- └── Continue test authoring

3.2.3 Module Editing & Versioning

- 1. Edit module from Modules page
- ├── Open existing module
- ├── Current version displayed (e.g., v1, v2)
- └── Edit mode enabled
-
- 2. Make changes
- ├── Modify test steps:
- │   ├── Edit step description
- │   ├── Edit expected outcome
- │   ├── Add new steps
- │   ├── Delete steps
- │   └── Reorder steps
- ├── Update module metadata:
- │   ├── Name
- │   ├── Description
- │   ├── Tags
- │   └── Linked projects
- └── For KaneAI-authored modules:
- └── Cannot modify advanced step types (variables, API, etc.) from UI
- └── Must edit via KaneAI agent session

-
- 3. Save changes
-     ├── New version created (e.g., v1 → v2)
-     ├── Version history maintained
-     ├── Previous versions accessible
-     └── Timestamp and user recorded
-
- 4. Impact on test cases
-     ├── Test cases using this module:
-     │   ├── Continue using current version by default
-     │   ├── Option to update to latest version
-     │   └── Impact analysis shows affected test cases
-     └── Version update workflow:
-         ├── View test cases using module
-         ├── Select test cases to update
-         ├── Apply latest version
-         └── Test cases updated with new module version

3.2.4 Module Editing from KaneAI Agent

- 1. During KaneAI session in Draft state
-     ├── Agent paused
-     └── Module step visible in test case
-
- 2. Expand module step
-     ├── Click to view module details

- ├── Shows all steps within module
- └── Shows module version
-
- 3. Edit module steps
- ├── Click "Edit Module" button
- ├── Modify existing steps:
- │ ├── Edit step instructions
- │ ├── Edit expected outcomes
- │ └── Update step parameters
- ├── Add new steps between existing ones
- ├── Delete steps
- └── Reorder steps
-
- 4. Save module changes
- ├── New module version created automatically
- ├── Version incremented (e.g., v2 → v3)
- ├── Test case now references new version
- └── Other test cases using older version unaffected
-
- 5. Resume agent
- ├── Agent continues with updated module
- └── Module changes reflected in execution

3.2.5 Module Reference & Execution

- 1. Reference module in test case

```
•     ├── During test authoring (KaneAI or Manual)
•     ├── Add Module step
•     ├── Select module from list:
•     │   ├── Filter by linked projects
•     │   ├── Search by name or tags
•     │   └── View module details
•     └── Select specific version (default: latest)
•
•  2. Module step in test case
•     ├── Displays as single step
•     ├── Shows module name and version
•     ├── Can expand to view contained steps
•     └── Cannot modify steps inline (must edit module)
•
•  3. Execution behavior
•     ├── During test execution:
•     │   ├── Module step expands to individual steps
•     │   ├── Each step executed sequentially
•     │   ├── Variables and parameters resolved
•     │   ├── Secrets fetched securely
•     │   ├── API/DB/JS steps executed
•     │   └── Expected outcomes validated (for Manual Test Steps)
•     ├── Failure handling:
•     │   ├── Step failure within module follows failure condition
•     │   ├── Module step fails if any contained step fails
```

- `│   └── Execution continues based on module step failure condition`
- `└── Reporting:`
- `    ├── Module step results aggregated`
- `    ├── Individual step results available in details`
- `    └── Module version recorded in results`

3.3 Test Execution Workflow

Learn more:

- KaneAI HyperExecute Test Run Execution
- Test Run Configurations
- Scheduled Test Runs in KaneAI
- KaneAI CI/CD Automation

3.3.1 KaneAI Generated Test Run Execution (Automated)

- `1. Test Run Creation`
- `├── Type: KaneAI Generated`
- `├── Select KaneAI authored test cases from repository`
- `│   └── Only test cases with type "KaneAI Authored" allowed`
- `├── Configure execution settings:`
- `│   ├── Browser/Device matrix`
- `│   ├── Parallel execution count (concurrency)`
- `│   ├── Environment selection`
- `│   ├── Region selection`
- `│   ├── Network throttling`
- `│   ├── Tunnel configuration`

- │   ├── Mobile app ID override
- │   └── Retry mechanisms
- └── Schedule or execute immediately
-
- 2. Test Run State: Not Started → In Progress
- ├── HyperExecute Job Creation:
- │   ├── API call to HyperExecute service
- │   ├── Resource allocation (VMs/devices/browsers)
- │   ├── Deploy generated automation code
- │   ├── Apply environment variables
- │   └── Execution initialization
-
- 3. Test Instance Execution (Parallel)
- ├── For each test case in run:
- │   ├── Create test instance (test case + configuration)
- │   ├── Apply browser/device configuration
- │   ├── Execute generated code steps sequentially:
- │   │   ├── Action steps
- │   │   ├── Assertion steps
- │   │   ├── API steps
- │   │   ├── Database steps
- │   │   ├── Module steps
- │   │   └── JavaScript steps
- │   ├── Handle failures per step condition:
- │   │   ├── Fail immediately → Stop test instance

- │   │   ├── Fail but continue → Log and continue
- │   │   └── Warn but continue → Warning and continue
- │   ├── Collect artifacts:
- │   │   ├── Screenshots
- │   │   ├── Videos
- │   │   ├── Logs
- │   │   └── Network traces
- │   └── Update instance status (Passed/Failed/Skipped)
- └── Aggregate results across all instances
-
- 4. Test Run State: In Progress → Passed/Failed
- ├── All instances passed → Passed
- └── One or more failed → Failed
-
- 5. Results Processing
- ├── Update test run status
- ├── Generate execution reports
- ├── Trigger notifications (email, Slack, webhooks)
- ├── Update dashboards and insights
- ├── Sync results to integrated systems (Jira, ADO)
- └── Archive artifacts in object storage
-
- 6. Retry Flow (Optional)
- └── Failed runs can be re-executed
-     └── Failed → Retry → In Progress

3.3.2 Non-KaneAI Generated Test Run Execution (Manual Tracking)

- 1. Test Run Creation
- ├── Type: Non-KaneAI Generated
- ├── Select non-KaneAI test cases from repository:
- │ ├── Manual Test Steps
- │ └── BDD Scenarios
- ├── Configure test run metadata:
- │ ├── Name, description
- │ ├── Assigned testers
- │ ├── Environment information
- │ └── Custom fields
- └── Test Run State: Custom status (defined in organization settings)
-
- 2. Manual Test Execution
- ├── Testers execute tests manually
- ├── Follow test case steps:
- │ ├── Manual Test Steps: Step-by-step instructions
- │ └── BDD Scenarios: Given-When-Then scenarios
- ├── Manually record results:
- │ ├── Mark steps as Passed/Failed
- │ ├── Add comments and observations
- │ ├── Attach screenshots/evidence

- │   └── Log defects
- └── No HyperExecute job creation (manual execution only)
-
- 3. Status Management
- ├── Custom statuses update manually:
- │   ├── Via Test Manager UI
- │   ├── Via API integration
- │   └── Via workflow automation
- └── Status transitions follow organization-defined rules
-
- 4. Results Tracking
- ├── Manual status updates tracked
- ├── Generate manual execution reports
- ├── Track test coverage
- ├── Link to issues/defects
- └── Export results (CSV, PDF)

3.4 AI Test Case Generator Workflow

Phase 1: Generation Request

- 1. User initiates test generation
- ├── Selects target Project and Folder
- ├── Chooses input source:
- │   ├── Text description
- │   ├── Image/Screenshot upload

- `│   ├── Audio/Voice input`
- `│   ├── Document upload (PDF, Word)`
- `│   ├── Jira issue reference`
- `│   └── Azure DevOps work item reference`
- `└── Submits generation request`
-
- `2. Generate entity created`
- `├── Generate ID assigned`
- `├── Input type recorded`
- `├── Input content stored/referenced`
- `├── User and timestamp captured`
- `├── Processing status: Pending`
- `└── Linked to Project and Folder`

Phase 2: AI Processing & Test Scenario Generation

- `1. LLM Processing (Azure OpenAI)`
- `├── Input preprocessing and sanitization`
- `├── Context analysis and enrichment`
- `├── Scenario identification`
- `└── Test case generation`
-
- `2. Test Scenarios created (2-10 scenarios)`
- `├── Scenario categorization:`
- `│   ├── Must-have scenarios (critical)`
- `│   ├── Should-have scenarios (important)`

- │   └── Could-have scenarios (optional)
- ├── Test type assignment:
- │   ├── Functional tests (all scenarios except last)
- │   └── Non-Functional tests (last scenario only)
- │   └── Performance, Security, Accessibility, etc.
- └── Each scenario contains 2-10 test cases
-
- 3. Test Cases generated per scenario
- ├── Title
- ├── Description
- ├── Pre-conditions
- ├── Test Steps (manual format)
- ├── Expected Outcomes per step
- └── For Functional tests only:
- └── KaneAI Test Plan generated
- ├── Bifurcation prompt processing
- ├── Test steps converted to adaptive mode format
- ├── Indicates "Automation ready" status
- └── Prepared for agent-based authoring

Phase 3: User Review & Selection

- 1. Generated test cases presented to user
- ├── Grouped by scenarios
- ├── Categorized by Must/Should/Could-have
- ├── Shows test type (Functional/Non-Functional)

- ├── Displays test steps and expected outcomes
- └── Indicates "Automation ready" status
- (Functional tests with KaneAI Test Plan)
-
- 2. User reviews and selects test cases
- ├── Can review individual test case details
- ├── Can edit test case details before creation
- ├── Selects which test cases to keep
- └── Counts displayed:
- ├── Distinct test cases
- ├── Automation ready (with KaneAI Test Plan)
- └── Error in automation plan (needs resolution)

Phase 4: Test Case Creation - Two Flows

Flow A: Create Test Cases (Manual Test Cases)

- 1. User clicks "Create Test Cases"
- ├── Selected test cases created as Manual Test Steps type
- ├── Saved to target Project and Folder
- ├── Test Steps: Manual format (Step + Expected Outcome)
- ├── KaneAI Test Plan: Discarded (not used)
- ├── Initial Status: Unverified or Ready (based on settings)
- └── Test cases available in Test Manager
-
- 2. Test cases stored as Non-KaneAI type

- └── Can be executed via Non-KaneAI Generated Test Runs
- └── Manual execution tracking only

Flow B: Create & Automate (KaneAI Authored via Agent)

- 1. User clicks "Create & Automate"
- ├── Configure automation settings:
- │ ├── Test platform (Desktop Browser/Mobile Browser/Mobile App)
- │ ├── Application under test (URL)
- │ ├── Agent Concurrency (1-100)
- │ ├── Generate data dynamically (toggle)
- │ └── Dismiss pop-ups (toggle)
- └── Only "Automation ready" test cases proceed
-
- 2. Test cases created (initial version)
- ├── Type: Manual Test Steps (initially)
- ├── Saved to target Project and Folder
- ├── KaneAI Test Plan: Retained
- ├── Initial Status: Unverified
- └── Each test case assigned to automation queue
-
- 3. For each "Automation ready" test case:
- ├── Session created (Background Mode)
- │ ├── Session State: Queued
- │ ├── Execution Mode: Background
- │ ├── Trigger Mode: Edit Existing Test

- │   ├── Platform: From user settings
- │   └── Concurrency: Based on subscription + user selection
- │
- ├── Session → Running (when resources available)
- │   ├── Agent initialized
- │   ├── Test Case loaded
- │   ├── Agent State: Setting Up → Recording → Idle
- │   └── Agent reads KaneAI Test Plan
- │
- ├── Agent authoring (Background Mode - Adaptive)
- │   ├── Agent processes KaneAI Test Plan steps
- │   ├── Agent State: Idle → Authoring → Idle (repeat)
- │   ├── Steps executed in adaptive mode
- │   ├── Variables/Parameters created as needed
- │   └── Handles errors automatically (retry or mark faulty)
- │
- └── Authoring outcomes:
-     ├── Success:
-     │   ├── Agent completes all steps (no skipped/failed steps)
-     │   ├── Session → Completed
-     │   ├── Test Case: New version created
-     │   ├── Type changed: Manual → KaneAI Authored
-     │   ├── Status: Ready
-     │   └── Code Generation (automatic):
-     │   ├── Code State: Generating

- `│       ├── Code State: Validating`
- `│       ├── Code State: Success`
- `│       ├── Automation script generated`
- `│       └── Test case ready for execution via HyperExecute`
- `│`
- `└── Failure:`
- `    ├── Agent stuck in Error state OR any step skipped/failed`
- `    ├── Session → Completed (with failure)`
- `    ├── Test Case: Status → Faulty`
- `    ├── Type remains: Manual Test Steps`
- `    ├── No code generation (generation condition not met)`
- `    └── Error details logged`
-
- `4. Automation summary`
- `   ├── Successfully automated count (with generated code)`
- `   ├── Failed automation count (no code generated)`
- `   └── Manual test cases (non-automation ready) count`

3.5 Automate with KaneAI Workflow (Manual Test Case → KaneAI Authored)

This workflow allows users to convert existing Manual Test Steps type test cases into KaneAI Authored test cases with automation code.

Trigger

- User opens an existing Manual Test Steps type test case

- Clicks "Automate with KaneAI" button
- Available for test cases that have test steps and description

Workflow

- `1. User initiates "Automate with KaneAI"`
- `   ├── Source: Existing Manual Test Steps type test case`
- `   ├── Inputs sent to Bifurcation Prompt:`
- `   │   ├── Test case description`
- `   │   ├── Test steps (manual format)`
- `   │   └── Expected outcomes per step`
- `   └── Configure session settings (platform, URL, etc.)`
-
- `2. Bifurcation Prompt Processing`
- `   ├── IMPORTANT: All steps are classified as GENERATIVE (not deterministic)`
- `   │   └── Reason: Manual test steps lack element-specific details`
- `   ├── Generates KaneAI Test Plan from manual steps`
- `   ├── Each manual step converted to adaptive/generative instruction`
- `   └── Plan ready for agent execution`
-
- `3. Session Creation (Background Mode)`
- `   ├── Session State: Queued → Running`
- `   ├── Execution Mode: Background`
- `   ├── Trigger Mode: Edit Existing Test`
- `   └── Agent initialized with KaneAI Test Plan`
-

- 4\. Agent Authoring (Generative/Adaptive Mode)
- `    ├── For each step in KaneAI Test Plan:`
- `    │   ├── Generative prompt analyzes current screen`
- `    │   ├── Determines atomic actions to achieve step goal`
- `    │   ├── Executes actions via Actor prompt`
- `    │   └── Loops until step goal achieved`
- `    ├── AI explores UI to figure out correct elements`
- `    ├── Creates variables/parameters as needed`
- `    └── Handles dynamic content adaptively`
-
- 5\. Authoring Outcomes
- `    ├── Success:`
- `    │   ├── All steps completed`
- `    │   ├── Test case type changed: Manual → KaneAI Authored`
- `    │   ├── New version created with authored steps`
- `    │   ├── Code generation triggered automatically`
- `    │   └── Test case ready for automated execution`
- `    │`
- `    └── Failure:`
- `        ├── Agent stuck or steps failed`
- `        ├── Test case status: Faulty`
- `        ├── Type remains: Manual Test Steps`
- `        └── Error details available for review`

Key Difference from "Create & Automate"

- **Create & Automate**: Starts from AI-generated test cases (from Generate entity)
- **Automate with KaneAI**: Starts from existing manual test cases created by users

Why Generative Mode First

When automating manual test cases, the bifurcation prompt classifies all steps as **generative** because:

1. Manual test steps describe "what to do" not "how to do it"
2. Steps lack element-specific details (no XPath, CSS selectors)
3. AI needs to explore the UI to find correct elements
4. Same manual step may require different UI interactions on different apps

Example:

- Manual step: "Enter username and password and click login"
- This becomes a generative instruction where AI figures out:
    - Where is the username field?
    - Where is the password field?
    - Where is the login button?
    - What actions to perform?

3.6 Integration Workflows

Jira Integration Flow

- `1. Issue Creation in Jira`
- `2. LambdaTest Jira App detects issue`
- `3. User triggers test generation`
- `4. AI analyzes issue content`
- `5. Test cases generated and linked`
- `6. Execution results sync back to Jira`
- `7. Issue status updated based on results`

GitHub Integration Flow

- `1. Developer creates Pull Request`
- `2. Comments "@LambdaTest validate"`
- `3. KaneAI analyzes code changes`
- `4. Generates relevant test cases`
- `5. Executes tests via HyperExecute`
- `6. Posts results with AI-powered RCA`
- `7. Provides merge recommendation`

Data Models & Structures

4.1 Test Case Data Model

- `{`
- `  "testCaseId": "TC-12345",`
- `  "projectId": "PROJ-001",`
- `  "folderId": "FOLD-123",`
- `  "title": "User Login Test",`
- `  "description": "Validates user login functionality",`
- `  "type": "Functional",`
- `  "priority": "High",`
- `  "status": "Approved",`
- `  "version": "2.1",`
- `  "preConditions": "User account exists",`
- `  "tags": ["login", "authentication", "critical"],`
- `  "customFields": {`
- `    "component": "Authentication",`

- `    "automationStatus": "Automated"`
- `  },`
- `  "steps": [`
- `    {`
- `      "stepNumber": 1,`
- `      "type": "action",`
- `      "instruction": "Navigate to login page",`
- `      "expectedResult": "Login page loads",`
- `      "failureCondition": "fail_immediately",`
- `      "timeout": 30000`
- `    }`
- `  ],`
- `  "linkedIssues": ["JIRA-123", "ADO-456"],`
- `  "createdBy": "user@example.com",`
- `  "createdAt": "2024-01-01T00:00:00Z",`
- `  "modifiedAt": "2024-01-15T00:00:00Z"`
- `}`

4.2 Test Run Configuration Model

- `{`
- `  "testRunId": "TR-98765",`
- `  "name": "Regression Test Suite",`
- `  "projectId": "PROJ-001",`
- `  "testCaseIds": ["TC-12345", "TC-12346"],`

```
  "configuration": {
    "platform": "web",
    "browsers": ["chrome", "firefox"],
    "os": ["Windows 10", "macOS"],
    "concurrency": 5,
    "environment": "staging",
    "tunnel": "my-tunnel",
    "networkThrottle": "3G",
    "geolocation": "US",
    "timezone": "America/New_York",
    "retryOnFailure": true,
    "maxRetries": 2
  },
  "schedule": {
    "type": "recurring",
    "cron": "0 0 * * *",
    "timezone": "UTC"
  },
  "notifications": {
    "email": ["team@example.com"],
    "slack": "#test-results"
  }
}
```

4.3 Variable Structure

```
{
  "variables": {
    "local": {
      "username": "testuser",
      "tempValue": "12345"
    },
    "global": {
      "baseUrl": "https://app.example.com",
      "apiKey": "{{secrets.org.api_key}}"
    },
    "environment": {
      "staging": {
        "dbConnection": "staging.db.example.com",
        "apiEndpoint": "https://api-staging.example.com"
      },
      "production": {
        "dbConnection": "prod.db.example.com",
        "apiEndpoint": "https://api.example.com"
      }
    },
    "smart": {
      "timestamp": "{{smart.current_timestamp}}",
      "randomEmail": "{{smart.random_email}}",
      "sessionId": "{{smart.session_id}}"
    }
```

- }
- }

KaneAI Core Features

Documentation:

- Getting Started with KaneAI
- Why We Need KaneAI
- Author Your First Desktop Browser Test
- Author Your First Mobile Browser Test
- Author Your First Mobile App Test

1. Test Authoring Capabilities

Learn more: Web Test Writing Guidelines

Natural Language Instructions

- Write tests in plain English
- AI interprets and converts instructions to executable steps
- Supports both structured and adaptive authoring modes

Manual Interaction Mode

Learn more: Manual Interaction in KaneAI

- Record user actions as test steps
- Click, type, navigate, and scroll actions captured
- Toggle between manual and AI-guided modes seamlessly

Hybrid Authoring

- Combine AI suggestions with manual recordings
- Best of both worlds: speed and precision
- Switch between modes during same session

Multi-Tab/Window Support

- Test complex workflows across browser contexts
- Handle pop-ups, new windows, and multiple tabs
- Maintain context across different browser instances

Drag & Drop Support

Learn more: Drag & Drop in KaneAI

- Visual element interaction testing
- Supports drag-and-drop UI components
- Validates drag-and-drop functionality

2. Advanced Testing Features

API Testing

Learn more: API Testing in KaneAI

- REST API testing with validation
- Support for all HTTP methods (GET, POST, PUT, DELETE, etc.)
- Request/response validation
- Header and parameter configuration
- Authentication support

Database Testing

Learn more: Database Testing in KaneAI

- SQL/NoSQL query execution and validation
- Support for multiple database types
- Result set validation
- Data-driven testing from database

Network Assertions

Learn more: Network Assertions in KaneAI

- Validate network calls, headers, and responses
- Monitor API calls during test execution
- Assert on network traffic patterns
- Capture and validate webhooks

JavaScript Execution

Learn more: JavaScript Execution in KaneAI

- Custom JS snippets within tests
- Execute browser-side JavaScript
- Manipulate DOM elements
- Advanced scripting capabilities

File Upload/Download

Learn more: File Upload and Download in KaneAI

- Handle file operations in tests
- Upload files to web applications
- Download and validate files
- Support for multiple file formats

TOTP Authentication

Learn more: TOTP Authentication in KaneAI

- Built-in 2FA/MFA support
- Time-based one-time password generation
- Seamless authentication flow testing
- No external authenticator apps needed

Bug Suggestion Generation

When test authoring is performed using generative/adaptive mode and an assertion fails, KaneAI automatically generates a bug suggestion to streamline defect reporting.

How It Works

1. **Trigger**: Assertion fails during generative step execution
2. **AI Analysis**: System analyzes the failure context
3. **Bug Suggestion Generated**: Includes:
    - **Actual Outcome**: What actually happened (captured from execution)
    - **Expected Outcome**: What was expected (from assertion definition)
    - **Screenshot**: Visual evidence at the point of failure
    - **Step Context**: The instruction and surrounding steps
    - **Environment Details**: Browser/device, URL, timestamp

Bug Suggestion Contents

- `Bug Suggestion`
- `├── Title: Auto-generated descriptive title`
- `├── Actual Outcome: [What happened]`
- `├── Expected Outcome: [What should have happened]`
- `├── Screenshot: [Captured at failure point]`
- `├── Steps to Reproduce: [From test steps]`
- `├── Environment: [Browser, OS, device details]`
- `└── Severity Suggestion: [Based on assertion type]`

Integration with Issue Trackers

- **Direct Bug Creation**: Create bug directly in integrated tools:
  - Jira
  - Azure DevOps
  - Other supported project management tools
- **Pre-filled Fields**: Bug details auto-populated from suggestion
- **Attachment Support**: Screenshot automatically attached
- **Bidirectional Linking**: Bug linked back to test case

Use Cases

- Rapid defect logging during exploratory testing
- Automated bug reporting in CI/CD pipelines
- Consistent bug documentation across team
- Reduced manual effort in defect reporting

3. Mobile Testing Capabilities

Learn more: Mobile App Test Writing Guidelines

Native App Testing

Learn more: Mobile App Capabilities in KaneAI

- Android and iOS app testing
- Real device cloud access
- Native element interaction
- App-specific gestures and actions

Mobile Browser Testing

- Responsive web testing on real devices
- Mobile Safari and Chrome support
- Touch gestures and mobile interactions
- Viewport and orientation testing

Advanced Mobile Features

Biometric Authentication Testing

- Fingerprint testing
- Face ID testing
- Biometric simulation on real devices

Camera Image Injection

- Mock camera input with images
- Test camera-dependent features
- Upload custom images as camera feed

Video Injection

- Mock video input
- Test video capture features
- Upload custom videos as camera feed

Screenshot Block Bypass

- Test apps with screenshot restrictions
- Capture screenshots for debugging
- Bypass security restrictions for testing

Deeplink Support

Learn more: Deeplink Support in KaneAI

- Test deeplink handling

- Validate deeplink navigation
- Cross-app navigation testing

Geolocation Testing

Learn more: Geolocation, Tunnel, and Proxy in KaneAI

- GPS mocking
- Location-based testing on mobile devices
- Test location-dependent features
- Change GPS coordinates during test

4. Test Organization & Reusability

Modules

Learn more:

- Modules in KaneAI
- Module Versions and Enhancements

Modules are reusable collections of test steps that can be shared across both KaneAI-authored and Manual Test Steps type test cases. They enable efficient test maintenance by centralizing common workflows.

Module Characteristics

- Reusable collection of test steps
- Share common workflows across multiple test cases
- Can be associated with multiple projects
- Project-level and organization-level modules
- Cannot nest modules (modules cannot reference other modules)
- Each module contains test steps with expected outcomes per step
- Expected outcomes are irrelevant for KaneAI test cases (only used in Manual Test Steps)

Module Creation Methods

1. Direct Creation from Modules Page

- Created at `kaneai.lambdatest.com/modules`
- Modules not yet authored in KaneAI
- Contains only basic test steps and expected outcomes in string format
- Cannot contain API, variables, parameters, DB, or JS type steps

- Compatible with both Manual Test Steps and KaneAI test cases

2. Creation from KaneAI Agent

- Created during KaneAI session by pausing agent
- Select consecutive test steps (authored or in queue) to convert to module
- Can include advanced step types:
    - Variables and parameters
    - Secrets (e.g., `{{secrets.org.lt_cloud_ops_password}}`)
    - API calls
    - Database queries
    - JavaScript execution
- When agent is paused, users can:
    - Edit module steps
    - Add steps in between
    - These edits create a new module version

Module Versioning

- Track changes to modules over time
- Maintain multiple versions
- Version updates occur when:
    - Test steps are modified
    - Expected outcomes are changed
    - Other module fields are updated
- Roll back to previous versions
- Impact analysis for module changes
- Update test cases to latest module version

Variables & Parameters

Learn more:

- Variables and Parameters in KaneAI
- Using Variables in KaneAI
- Using Parameters in KaneAI
- Smart Variables in KaneAI
- Using Datasets in KaneAI

Variable Types

- **Local Variables**: Test-specific, session-bound
- **Global Variables**: Organization-wide access
- **Environment Variables**: Environment-specific values
- **Smart Variables**: System-generated dynamic values
- **Parameters**: Runtime configuration values

Variable Scopes

- Local (test-level)
- Global (organization-level)
- Environment-specific
- Parameter-based (data-driven testing)

Variable Usage

- Reference with `{{variable_name}}` syntax
- Can reference other variables
- Persist flag for local to global promotion
- Dynamic value substitution

Environments

- Multi-environment test execution
- Environment-specific variable mappings
- Configuration overrides per environment
- Seamless environment switching
- Staging, production, and custom environments

Secrets Management

Learn more: Secrets in KaneAI

- Secure credential storage using HashiCorp Vault
- User-level and organization-level secrets
- Encrypted at rest (AES-256)
- Reference with `{{secrets.x}}` syntax
- Audit trail for secret access
- Never exposed in logs or reports

Smart Variables

- Context-aware dynamic values
- System-generated values
- Common smart variables:
  - `{{smart.current_timestamp}}`
  - `{{smart.random_email}}`
  - `{{smart.session_id}}`
  - `{{smart.random_string}}`
  - `{{smart.uuid}}`

Datasets (Data-Driven Testing)

- Collection of structured test data
- Default dataset (auto-generated, immutable)
- Custom datasets (user-created, editable)
- Support for:
    - Manual data entry
    - AI Autofill
    - CSV Import
- Parameters as columns
- Multiple rows for different test scenarios

5. Test Creation Workflows

Session Configuration Capabilities

When starting a KaneAI authoring session, users can configure various capabilities that affect test execution environment.

Network & Connectivity (All Platforms)

Tunnel Support

Learn more: Geolocation, Tunnel, and Proxy in KaneAI

- Connect to private/internal applications
- Access localhost and staging environments
- Secure tunnel for internal network testing
- Required for database connections to private databases
- Configure tunnel name in session settings

Geolocation Testing

- GPS mocking for mobile devices
- IP-based geolocation for web testing
- Test location-dependent features
- Change location during authoring session
- Supports multiple regions worldwide

Dedicated Proxy

- Route traffic through dedicated proxy servers
- Useful for testing behind corporate firewalls
- Custom proxy configuration per session
- Supports authenticated proxies

Web Authoring Specific Capabilities

Custom Headers

Learn more: Custom Headers in KaneAI

- Inject custom HTTP headers in all requests
- Use cases:
    - Authentication headers (Bearer tokens, API keys)
    - Custom user agent strings
    - Feature flag headers
    - A/B testing headers
    - Debug/trace headers
- Configure before session starts
- Headers applied to all requests during session

Chrome Options (Arguments)

Learn more: Chrome Options in KaneAI

- Pass custom Chrome command-line arguments
- Common use cases:
    - `--disable-web-security`: Disable CORS (for testing)
    - `--start-maximized`: Start browser maximized
    - `--disable-notifications`: Block notification prompts
    - `--disable-popup-blocking`: Allow popups
    - `--ignore-certificate-errors`: Ignore SSL errors
    - Custom experimental features
- Configure via session settings before launch
- Only available for Chrome/Chromium-based browsers

Mobile Authoring Specific Capabilities

Device Type Selection

- **Public Devices**: Shared device pool (see Device Cloud documentation)
- **Private Devices**: Dedicated devices for your account only

App Configuration

- Upload APK (Android) or IPA (iOS) files
- Select from previously uploaded apps
- App versioning support

Mobile-Specific Features

- Biometric simulation (fingerprint, Face ID)
- Camera image/video injection

- GPS mocking
- Deeplink testing

Configuration Summary by Platform

Capability	**Web Desktop**	**Web Mobile**	**Mobile App**
Tunnel	Yes	Yes	Yes
Geolocation (GPS)	No	Yes	Yes
IP Geolocation	Yes	Yes	Yes
Dedicated Proxy	Yes	Yes	Yes
Custom Headers	Yes	No	No
Chrome Options	Yes (Chrome)	No	No
Network Throttling	Yes	Yes	Yes
Public/Private Devices	N/A	Yes	Yes

KaneAI Authoring Flow

1. **Initiation**: Author Web/App test
2. **Configuration**: Select browser/device, set capabilities (tunnel, proxy, headers, etc.)
3. **Authoring**:
   - Natural language instructions
   - Manual interactions
   - Slash commands for advanced actions
4. **Execution**: Real-time test step processing
5. **Validation**: Assertions and verifications
6. **Save**: Store in Test Manager with metadata

Slash Commands

Learn more: KaneAI Command Guide

Special commands for advanced actions:

- `/api` - API testing

- `/js` - JavaScript execution
- `/db` - Database queries
- `/network` - Network assertions
- `/totp` - TOTP authentication
- `/upload` - File upload
- `/download` - File download
- `/variable` - Create/manage variables
- `/secret` - Create/manage secrets
- `/parameter` - Create/manage parameters

6. Variable & Data Management

Variable Lifecycle

1. **Creation**: Via UI, natural language, or API responses
2. **Usage**: `{{variable_name}}` syntax in test steps
3. **Persistence**: Optional value persistence across sessions
4. **Environment Switching**: Automatic value substitution based on selected environment

Parameter Management

- Define parameters for data-driven testing
- Link parameters to datasets
- Multiple data rows for different test scenarios
- Parameter values injected during test execution

Secret Management Workflow

1. **Creation**: Add secrets via UI or API
2. **Storage**: Encrypted in HashiCorp Vault
3. **Access Control**: User-level or organization-level
4. **Usage**: Reference in tests with `{{secrets.key}}` syntax
5. **Audit**: All access logged and tracked

Test Manager Core Features

Documentation: For complete Test Manager documentation, visit LambdaTest Test Manager

1. Test Case Management

Learn more: Creating Projects in Test Manager

Repository Management

- Centralized test case storage
- Hierarchical organization with projects and folders
- Version control for test cases
- Test case metadata tracking
- Search and filter capabilities
- Tag-based organization

Folder Organization

- Hierarchical test organization
- Two separate folder types:
  - Test Case Folders
  - Test Run Folders
- Nested folder structure (up to N levels)
- Drag-and-drop folder management
- Bulk move operations

Test Case Types

- **KaneAI Authored**: AI-generated test cases with automation scripts
- **Manual Test Steps**: Traditional test cases with step-by-step instructions
- **BDD Scenarios**: Behavior-Driven Development test cases

Custom Fields

Learn more: System and Custom Fields

- Extensible test case metadata
- Organization-defined custom fields
- Support for various data types (text, dropdown, date, etc.)
- Custom field filtering and reporting

Import/Export

Learn more:

- CSV Import for Test Cases
- One-Click Migration from TestRail
- **Import**: CSV import for bulk test case creation
- **Export**: Multiple format exports
    - CSV
    - PDF
    - Excel
    - JSON
- Bulk operations support

Bulk Operations

- Mass updates and modifications
- Bulk status changes
- Bulk tagging
- Bulk delete/archive
- Bulk move to folders

Version History

- Track all changes to test cases
- View previous versions
- Compare versions
- Restore previous versions
- Audit trail for changes

2. Test Execution Management

Learn more:

- Test Run Creation and Management
- Automated Test Cases Linking (Dashboard)
- Automated Test Cases Linking (Capability)

Test Runs

- Organized test execution campaigns
- Two types:
    - **KaneAI Generated**: Automated execution via HyperExecute
    - **Non-KaneAI Generated**: Manual execution tracking
- Configuration options:
    - Browser/Device matrix
    - Parallel execution (concurrency)

    - Environment selection
    - Network throttling
    - Retry mechanisms

Test Plans

- Structured testing strategies
- Group related test runs
- Track testing progress
- Milestone association

Build Management

- Version-specific test execution
- Track test results per build
- Build comparison and analysis
- Release quality tracking

Parallel Execution

- Concurrent test running
- Configurable concurrency levels
- Resource allocation optimization
- Faster feedback cycles

Scheduled Executions

- Automated test scheduling
- Recurring schedules (cron-based)
- Timezone configuration
- Pre-scheduled test runs
- CI/CD integration

Test Execution Workflow

Manual Test Execution Flow

1. **Test Run Creation**: Select non-KaneAI test cases
2. **Configuration**: Set up test run metadata
3. **Assignment**: Assign testers
4. **Execution**: Testers execute tests manually
5. **Results**: Mark steps as Passed/Failed

6. **Reporting**: Generate execution reports

Automated Test Execution Flow

1. **Test Run Creation**: Select KaneAI authored test cases
2. **Configuration**: Set execution parameters
3. **Trigger**: Manual, scheduled, or API-triggered
4. **Execution**: HyperExecute orchestration
5. **Monitoring**: Real-time execution tracking
6. **Results**: Automated pass/fail status and artifacts

3. AI Test Generation

Learn more:

- Generate Multiple Tests with AI
- KaneAI Test Plan

Multi-Input Support

Generate test cases from various sources:

- **Text**: Plain text descriptions
- **Images**: Screenshots and mockups
- **Audio**: Voice inputs
- **Documents**: PDFs, Word docs
- **Third-Party**: Jira issues, Azure DevOps work items

Scenario Generation

- Automatic test scenario creation
- 2-10 scenarios per generation request
- Scenario categorization:
    - Must-have (critical)
    - Should-have (important)
    - Could-have (optional)

Test Categorization

- **Must-have**: Critical scenarios that must be tested
- **Should-have**: Important scenarios recommended for testing
- **Could-have**: Optional scenarios for comprehensive coverage

Test Types

- **Functional Tests**: Generated for most scenarios
- **Non-Functional Tests**: Performance, security, accessibility (last scenario)
- **Positive Tests**: Happy path scenarios
- **Negative Tests**: Error and edge case scenarios
- **Edge Cases**: Boundary and unusual scenarios

Third-Party Integration

- Generate from Jira issues
- Generate from Azure DevOps work items
- Automatic linking to source tickets
- Bidirectional traceability

Test Generation Flow

1. **Input**: Select input source and provide content
2. **Processing**: AI analyzes and generates scenarios
3. **Review**: Review and edit generated test cases
4. **Selection**: Choose which test cases to create
5. **Creation**: Two options:
    - **Create Test Cases**: Save as manual test cases
    - **Create & Automate**: Automatically author with KaneAI agents

Memory Layer (AI Context Enhancement)

The Memory Layer provides contextual intelligence to the AI Test Case Generator, enabling more relevant and non-duplicate test case generation.

Memory Layer Components

1. Project-Level Instructions

- Custom instructions set at the project level
- Provide domain-specific context for test generation
- Examples:
    - "This is an e-commerce application with cart functionality"
    - "Always include accessibility test cases"
    - "Focus on mobile-first test scenarios"
- Applied to all test generations within the project

2. Organization-Level Instructions

- Custom instructions set at the organization level
- Provide company-wide testing standards and context
- Examples:
    - "All tests must include security validations"
    - "Follow WCAG 2.1 guidelines for accessibility"
    - "Include performance baseline assertions"
- Applied across all projects in the organization
- Project-level instructions can override or extend organization instructions

3. Memory Enhancement (Repository Context)

- **Purpose**: Generate non-duplicate test cases by learning from existing test repository
- **How it works**:
    - User enables "Memory Enhancement" toggle in AI Test Generator
    - System fetches existing test cases from the current project
    - Context is extracted from existing test cases:
        - Test case titles and descriptions
        - Test step patterns
        - Coverage areas already tested
    - LLM uses this context to:
        - Avoid generating duplicate test cases
        - Identify gaps in current test coverage
        - Generate complementary test scenarios
- **Benefits**:
    - Reduces duplicate test case creation
    - Improves test coverage completeness
    - Learns from existing testing patterns
    - Maintains consistency with existing test suite

Memory Layer Configuration

- `Organization Settings`
- `└── AI Instructions (organization-level context)`
-
- `Project Settings`
- `└── AI Instructions (project-level context)`
-
- `AI Test Generator UI`
- `├── Project Instructions (displayed/editable)`
- `├── Organization Instructions (displayed)`

- `└── Memory Enhancement Toggle`
- `    └── When enabled: Fetches existing test cases for context`

Best Practices

- Keep project instructions specific to the application under test
- Use organization instructions for company-wide standards
- Enable Memory Enhancement for mature projects with existing test suites
- Review and update instructions periodically as the application evolves

4. Milestones

Learn more: Milestone Creation and Management

Purpose

- Central organizational tool for grouping test runs
- Track feature launches and releases
- Monitor testing progress
- Aggregate results across test runs

Milestone States

- **Active**: Currently in progress
- **Completed**: Finished and marked as complete

Milestone Attributes

- Name (required)
- Description
- Tags
- Owner (assigned team member)
- Duration (expected timeframe)
- Attachments
- Associated test runs

Use Cases

- Release tracking

- Sprint testing
- Feature launch monitoring
- Regression testing campaigns

5. Reports & Analytics

Learn more:

- Insights Dashboard
- Test Manager Reports

Execution Reports

- Analyze test execution results
- Identify patterns and trends
- Pass/fail metrics
- Execution time analysis
- Flaky test identification
- Historical trend analysis

Traceability Reports

- Show alignment between test cases and requirements
- Link test cases to defects
- Requirements coverage
- Gap analysis
- Impact analysis

Report Types

- Execution Reports
- Traceability Reports
- Custom Reports (with filters)

Report Features

- **Filters**: Test runs, dates, metadata
- **Recurring Reports**: Automated email delivery
- **Recipients**: Configure email recipients
- **Schedule Frequency**: Daily, weekly, monthly
- **Export**: PDF, CSV, Excel

Report Configuration

- Name and description
- Report type selection
- Filter criteria:
    - Test runs
    - Date ranges
    - Test metadata
    - Issue metadata
    - Run metadata
- Milestone association
- Recurring schedule setup

6. Test Instance Management

Learn more: Test Instance Audit Logs

Test Instances

- Specific execution of test case + configuration
- Track individual test results
- Capture execution artifacts:
    - Screenshots
    - Videos
    - Logs
    - Network traces
- Step-level results
- Execution duration tracking
- Failure analysis

Test Instance Attributes

- Instance ID
- Test case reference
- Execution configuration
- Start/end times
- Duration
- Result status (Passed/Failed/Skipped)
- Artifacts and logs

7. Datasets

Purpose

- Manage structured data for data-driven testing
- Reuse input values across test cases
- Support multiple test scenarios with different data

Dataset Types

- **Default Dataset**: Auto-generated during test authoring (immutable)
- **Custom Dataset**: User-created or copied from default (editable)

Dataset Features

- Parameters (columns/placeholders)
- Rows (test scenarios with different data values)
- Data types (text, numbers, dates, etc.)
- Version history
- Creation methods:
  - Manual data entry
  - AI Autofill
  - CSV Import

Dataset Usage

- Link datasets to test cases
- Map parameters to dataset columns
- Execute test cases with different data rows
- Data-driven test execution

Integrations & Ecosystem

Development Tool Integrations

1. Jira Integration

Learn more:

- Link Jira Issues with Test Manager
- LambdaTest Jira App
- KaneAI Jira Integration

LambdaTest Jira App

- Native Jira marketplace app
- Available in Atlassian Marketplace
- Easy installation and configuration

Bidirectional Sync

- Test cases ↔ Jira issues
- Automatic updates

Features

- **Link test cases to issues**: Associate test cases with Jira tickets
- **View execution history**: See test results directly in Jira
- **Create test cases from Jira**: Generate test cases from issue descriptions
- **AI-powered test generation**: AI analyzes Jira tickets and generates test scenarios

Jira Integration Workflow

1. Issue Creation in Jira
2. LambdaTest Jira App detects issue
3. User triggers test generation
4. AI analyzes issue content (description, acceptance criteria, attachments)
5. Test cases generated and linked to Jira issue
6. Test execution triggered
7. Execution results sync back to Jira
8. Issue status updated based on test results

Use Cases

- Generate tests from bug reports
- Generate tests from user stories
- Track test coverage per issue
- Sync defects from failed tests
- Automated test creation for new features

2. Azure DevOps Integration

Learn more:

- Link ADO Issues with Test Manager
- LambdaTest Azure DevOps App

LambdaTest ADO App

- Azure DevOps marketplace app
- Native integration with ADO
- Seamless installation

Features

- **Link test cases to work items**: Associate test cases with ADO work items
- **Test execution tracking**: Track test execution status in ADO
- **AI test generation**: Generate tests from work items
- **Bidirectional traceability**: Full traceability between tests and requirements

Azure DevOps Workflow

1. Work item creation in Azure DevOps
2. LambdaTest ADO App detects work item
3. User triggers test generation
4. AI analyzes work item content
5. Test cases generated and linked
6. Execution results sync back to ADO
7. Work item status updated

Use Cases

- Generate tests from user stories
- Generate tests from bugs
- Requirements traceability
- Sprint testing automation
- Release quality tracking

3. GitHub Integration

Learn more: GitHub App Integration

LambdaTest AI Cloud GitHub App

Overview: The LambdaTest AI Cloud GitHub App is a native GitHub marketplace app that leverages KaneAI to automatically generate and execute end-to-end tests based on pull request changes. It integrates directly with GitHub workflows, eliminating the need for manual test case creation and context switching.

Installation & Setup

Installation Steps:

1. Visit GitHub Marketplace and search for "LambdaTest AI Cloud"
2. Select repository access (all repositories or selective)
3. Authorize the app installation
4. Verify installation in GitHub organization settings under "Installed GitHub Apps"

Configuration Requirements: Create `.lambdatest/config.yaml` in repository root:

- `project_id: <Test Manager project ID>`
- `folder_id: <Folder ID where tests will be organized>`
- `assignee: <User ID for test runs>`
- `environment_id: <Browser/device configuration ID>`
- `test_url: <Application testing URL>`

Prerequisites:

- LambdaTest Enterprise Account
- KaneAI enabled (14-day free trial for new signups)
- GitHub repository administrative access
- Proper configuration file in repository
- README.md in repository (significantly improves test quality by providing context)

Complete Workflow

Trigger Commands: Post a comment on any PR with either:

- `@LambdaTest Validate this PR`
- `@KaneAI Validate this PR`

Execution Pipeline:

1. **Code Analysis**:
    - AI examines PR changes and repository context
    - Analyzes README files to understand application purpose and business logic
    - Reviews PR description and commit messages
    - Identifies affected components and features
2. **Test Generation**:
    - KaneAI creates intelligent test scenarios understanding business logic
    - Performs semantic duplicate detection against existing tests in Test Manager
    - Generates tests covering multiple browser/device combinations
    - Creates contextually relevant test cases specific to code changes

3. **Test Authoring**:
    - Generated test cases convert to executable scripts
    - Automatic assertion and validation logic creation
    - Code generated for configured framework
4. **Cloud Execution**:
    - Tests run on HyperExecute infrastructure
    - Parallel execution across multiple browsers/devices
    - Real-time execution with comprehensive artifact capture
5. **AI-Powered Reporting**:
    - Comprehensive results with pass/fail status
    - AI Root Cause Analysis automatically diagnoses test failures
    - Analysis of logs, screenshots, and stack traces
    - Remediation recommendations
    - PR approval guidance based on results

Real-Time Progress Tracking

Dynamic PR Comments: The app posts real-time updates directly to the PR showing:

- Current execution phase (Analysis, Generation, Authoring, Execution, Reporting)
- Test inventory (number of tests generated)
- Live execution progress
- Individual test results as they complete
- Links to HyperExecute dashboard for detailed logs/videos
- Permanent audit trail in PR history

Progress Phases:

1. **Analyzing**: Code change analysis in progress
2. **Generating**: Creating test scenarios
3. **Authoring**: Converting scenarios to executable code
4. **Executing**: Running tests on HyperExecute
5. **Analyzing Results**: AI-powered RCA in progress
6. **Complete**: Final summary with recommendations

Advanced Features

Duplicate Detection:

- Semantic analysis of existing tests in Test Manager
- Prevents redundant test generation
- Identifies similar test scenarios
- Suggests consolidation opportunities

Intelligent Test Selection:

- Analyzes code changes to determine relevant tests

- Generates only necessary tests for PR scope
- Avoids over-generation of irrelevant tests
- Focuses on impacted functionality

Test Manager Integration:

- All generated tests automatically sync to Test Manager
- Organized in configured folder structure
- Linked to PR for traceability
- Available for future regression runs
- Centralized visibility across teams

Root Cause Analysis (RCA):

- Automatic failure diagnosis
- Stack trace analysis
- Screenshot and video correlation
- Network and console log analysis
- Specific remediation steps
- Merge approval recommendations

Reporting Capabilities

Executive Summary:

- Total tests generated and executed
- Pass/fail ratio
- Test coverage metrics
- Overall PR quality assessment
- Merge recommendation (Approve/Review/Block)

Detailed Results:

- Individual test case results with pass/fail status
- Execution time per test
- Browser/device combination results
- Screenshots for failures
- Video recordings of test execution
- Network logs and console errors

AI Root Cause Analysis:

- Specific failure identification (e.g., "Login button not found")
- Probable cause (e.g., "Element locator changed")
- Impact assessment (e.g., "Critical - blocks user login")
- Recommended actions (e.g., "Update element ID in code")
- Related code changes that may have caused failure

Use Cases

Primary Use Cases:

- Automated regression testing for every PR
- Pre-merge validation and quality gates
- Code change impact analysis
- Continuous testing in shift-left workflows
- Developer self-service testing
- Reducing QA bottlenecks

Advanced Scenarios:

- Feature branch testing before merge
- Hotfix validation in production branches
- Integration testing across microservices
- UI/UX change validation
- API endpoint testing triggered by backend changes

Best Practices

Repository Setup:

- Maintain comprehensive README.md with feature descriptions
- Include clear PR descriptions explaining changes
- Use semantic commit messages
- Keep configuration file updated with latest environment IDs

Test Management:

- Review generated tests periodically
- Remove or consolidate duplicate tests
- Update test data and environments as needed
- Link tests to requirements/issues in Test Manager

Workflow Optimization:

- Trigger validation early in PR lifecycle
- Review AI-generated tests before merging
- Use RCA insights to improve code quality
- Integrate with branch protection rules

Limitations & Considerations

Current Limitations:

- Requires LambdaTest Enterprise Account
- GitHub only (GitLab/Bitbucket not yet supported)

- Requires proper configuration file setup
- Test quality depends on README and PR description quality

Performance Considerations:

- Test generation time varies based on PR complexity (typically 3-10 minutes)
- Execution time depends on number of tests and environments
- Parallel execution optimizes total execution time
- Large PRs may generate more tests (consider breaking into smaller PRs)

CI/CD Integration

API-Based Execution

- RESTful API for test triggering
- Programmatic test run creation
- Real-time status updates
- Webhook notifications

Supported Platforms

- **Jenkins**: Plugin and API integration
- **GitHub Actions**: Workflow integration
- **GitLab CI**: Pipeline integration
- **CircleCI**: Orb and API integration
- **Travis CI**: Configuration-based integration
- **Bamboo**: API integration
- **TeamCity**: API integration
- **Azure Pipelines**: Task integration

Configuration Options

Execution Settings

- **Concurrency control**: Set parallel execution count
- **Environment selection**: Choose target environment
- **Dynamic URL replacement**: Override application URLs
- **Region selection**: Select execution region
- **Network throttling**: Simulate network conditions
- **Retry mechanisms**: Configure automatic retries
- **Mobile app override**: Specify mobile app versions

CI/CD Workflow

1. Code commit triggers CI/CD pipeline
2. Build and compile code
3. Trigger LambdaTest test run via API
4. Execute tests in parallel
5. Collect results and artifacts
6. Update build status
7. Send notifications
8. Gate deployment based on test results

API Integration Example

- `POST https://test-manager-api.lambdatest.com/api/atm/v1/hyperexecute`

Key Parameters:

- `test_run_id`: Unique test run identifier
- `concurrency`: Parallel execution count
- `environment`: Target environment
- `replaced_url`: Dynamic URL mapping
- `region`: Execution region (US, EU, APAC)
- `android_app_id`: Android app override
- `ios_app_id`: iOS app override
- `network_throttle`: Network profile (2G, 3G, 4G)
- `tunnel`: Tunnel name for local testing

Benefits

- Continuous testing in CI/CD pipeline
- Fast feedback on code changes
- Automated quality gates
- Parallel test execution
- Comprehensive test coverage
- Early defect detection

Webhook Support

- Real-time notifications
- Custom webhook endpoints
- Test execution events
- Result notifications

- Failure alerts

Notification Channels

- Email notifications
- Slack integration
- Microsoft Teams integration
- Custom webhooks
- SMS alerts (enterprise)

Third-Party Tool Ecosystem

Issue Tracking

- Jira (native integration)
- Azure DevOps (native integration)
- GitHub Issues
- GitLab Issues

Project Management

- Jira (boards, sprints)
- Azure DevOps (boards)
- Asana (via API)
- Monday.com (via API)

Communication

- Slack (notifications)
- Microsoft Teams (notifications)
- Email (SMTP)

Version Control

- GitHub (native app)
- GitLab (API)
- Bitbucket (API)
- Azure Repos (ADO integration)

Integration Best Practices

Setup

1. Install marketplace apps from respective platforms
2. Configure authentication (API tokens, OAuth)
3. Set up project mappings
4. Configure sync settings
5. Test integration with sample data

Usage

1. Link test cases to external issues/work items
2. Use AI generation from external sources
3. Set up automated test execution in CI/CD
4. Configure notification channels
5. Monitor sync status and logs

Maintenance

1. Regularly review integration logs
2. Update API tokens when expired
3. Monitor rate limits
4. Keep marketplace apps updated
5. Review and optimize sync frequency

Advanced Capabilities & Configuration

Network & Connectivity Features

Network Throttling

Learn more: Network Throttling in KaneAI

Profiles

- **2G**: 250 Kbps download, 50 Kbps upload
- **3G**: 750 Kbps download, 250 Kbps upload
- **4G**: 4 Mbps download, 3 Mbps upload

- **Custom bandwidth settings**: Define custom profiles

Application

- Web testing
- Mobile testing (both browser and native apps)

Configuration

- **Pre-session**: Set network profile before session starts
- **Mid-session**: Change network conditions during test execution
- **Dynamic switching**: Test under varying network conditions

Use Cases

- Test application performance under poor network
- Validate offline functionality
- Test progressive web apps (PWA)
- Simulate real-world network conditions

Tunnel Support

Purpose

- Access to private/internal applications
- Local testing of applications not publicly accessible
- Secure connection to internal networks

Database Connections

- Secure database access
- Test database queries on private databases
- Connect to internal databases

Configuration

- `--expose` flag for specific services
- Port forwarding
- Custom tunnel names
- Multiple tunnel support

Use Cases

- Test applications on local development environment
- Access staging environments behind VPN
- Test integrations with internal APIs
- Database testing on private databases

Geolocation Testing

GPS Mocking (Mobile Only)

- Location-based testing on mobile devices
- Mock GPS coordinates
- Test location-dependent features
- Change location during test execution

IP Geolocation

- Regional content testing
- Test geo-restricted features
- Validate location-based personalization
- Test CDN routing

Timezone Testing

- Time-zone specific scenarios
- Test time-sensitive features
- Validate timezone conversions
- Test scheduling functionality

Supported Locations

- United States
- Europe (multiple countries)
- Asia Pacific (multiple countries)
- Custom coordinates

Custom Configuration Options

Browser Capabilities (Desktop Only)

Chrome Options

Learn more: Chrome Options in KaneAI

- Custom browser flags
- Experimental features
- Performance settings
- Security settings

Custom Headers

Learn more: Custom Headers in KaneAI

- Request header injection
- Authentication headers
- Custom user agents
- API keys in headers

Console/Network Logs

- Detailed debugging information
- JavaScript console logs
- Network request/response logs
- Performance metrics

Additional Options

- Browser extensions
- Download settings
- Notification permissions
- Camera/microphone permissions

Mobile Capabilities

Device Selection

- Real device cloud access
- Thousands of device/OS combinations
- Latest and legacy devices
- Popular device models

Public vs Private Devices

Public Devices:

- Common pool of devices shared across all LambdaTest users
- Device allocation is dynamic (first-come, first-served)
- Automatic cleanup after each session:

    - App data cleared
    - Browser cache cleared
    - Device reset to clean state
- Device deallocated and returned to pool after use
- Available for both authoring and test runs
- Cost-effective option for most testing scenarios

Private Devices:

- Dedicated devices reserved exclusively for a specific LambdaTest account
- No other accounts can access these devices
- Benefits:
    - Guaranteed availability (no waiting in queue)
    - Persistent device state (optional limited cleanup)
    - Consistent device configuration
    - Ideal for long-running tests or specialized configurations
- Limited cleanup options:
    - Can retain certain app data between sessions
    - Custom device configurations preserved
- Higher cost but guaranteed access
- Requires enterprise plan or special arrangement

Use Cases:

Scenario	Recommended
Standard test authoring	Public devices
CI/CD automated runs	Public devices
Sensitive data testing	Private devices
Long-running performance tests	Private devices
Specialized device configurations	Private devices
High-frequency testing with same device	Private devices

App Upload

- APK file management (Android)
- IPA file management (iOS)
- App versioning
- Private app storage

App Profiling

- Performance metrics
- CPU usage
- Memory usage
- Battery consumption
- Network usage

Biometrics

- Fingerprint testing
- Face ID testing
- Biometric authentication simulation
- Real device biometric sensors

Image/Video Injection

- Upload media to mock camera
- Test camera-dependent features
- Image capture testing
- Video recording testing

Additional Capabilities

- Push notifications
- SMS testing
- Phone calls (simulation)
- Device rotation
- Multi-touch gestures

Automation Code Generation

Learn more: Automation Code Generation in KaneAI

Supported Frameworks

Web Frameworks

Selenium:

- Python (PyTest) - **Available**
- Java - Coming soon
- C# - Coming soon

- Ruby - Coming soon

Playwright:

- JavaScript - Coming soon
- Python - Coming soon
- TypeScript - Coming soon

Cypress:

- JavaScript - Coming soon

WebdriverIO:

- JavaScript - Coming soon

Mobile Frameworks

Appium:

- Python (PyTest) - **Available**
- Java - Coming soon
- JavaScript - Coming soon
- C# - Coming soon

Code Generation Features

Multi-language Support

- Support for multiple programming languages
- Framework-specific code generation
- Best practices for each framework

Framework-specific Optimizations

- Follows framework conventions
- Uses framework best practices
- Optimized for performance
- Maintainable code structure

Customizable Templates

- Modify generated code structure
- Custom naming conventions
- Organization-specific patterns
- Configurable code style

Direct Execution

- Run generated code via HyperExecute
- No manual setup required
- Automatic dependency management
- Pre-configured execution environment

Code Generation Workflow

1. **Test Authoring**: Complete test authoring in KaneAI
2. **Trigger**: Automatic trigger on successful authoring
3. **Generation**: Code generated for selected framework
4. **Validation**: Code validated for syntax and structure
5. **Storage**: Code stored with test case
6. **Execution**: Code executed via HyperExecute

Code Generation States

- **Generating**: Code generation in progress
- **Validating**: Generated code being validated
- **Success**: Code generated and validated successfully
- **Failure**: Code generation or validation failed

Auto-Heal Feature

Learn more: Auto-Heal in KaneAI

Overview

Auto-Heal is an intelligent reliability mechanism that maintains test script robustness when application UIs change. It dynamically finds new locators at runtime by leveraging original natural language instructions, ensuring tests continue to execute successfully despite UI modifications.

How It Works

Multi-Locator Fallback System:

- Every element identified by KaneAI includes multiple locators (XPath, CSS, ID)
- During execution, if the primary locator fails, the system automatically attempts fallback locators in sequence
- Locators are prioritized based on confidence scores and stability

Natural Language Recovery:

- When all locators fail, the system re-evaluates the original natural language instruction
- AI analyzes the step's intent (e.g., "Click the Submit button")
- Rebuilds the locator based on contextual understanding and current page structure
- Uses DOM analysis considering page structure and nearby elements
- Integrates visual query when DOM-based detection fails

Activation & Scope

Automatic Activation:

- Auto-Heal activates automatically during HyperExecute test runs if test is authored in KaneAI
- Triggers when element locators break or become invalid
- No manual configuration required

Supported Commands: Any command that requires element locators falls under Auto-Heal:

- Click
- Type/Input
- Hover
- Select
- Assert
- Any interaction requiring element identification

Execution Behavior

During Test Runs:

- Tests continue execution without interruption when locators fail
- System attempts multiple recovery strategies
- Maintains test execution speed with minimal overhead
- Logs all healing activities for transparency

Locator Handling Strategies:

1. **Fallback Locators**: Try alternative pre-generated locators
2. **Contextual DOM Analysis**: Analyze page structure and nearby elements
3. **Visual Query Integration**: Use visual recognition when DOM methods fail
4. **Natural Language Re-evaluation**: Rebuild locator from original intent

Logging & Reporting

Current Capabilities:

- Healing events are logged during execution
- Test results indicate which steps were auto-healed

- Available in HyperExecute execution logs

Future Enhancements (Roadmap):

- Detailed healing reports
- Automatic locator updates in regenerated scripts
- Healing analytics and patterns
- Configurable healing strategies

Best Practices

Optimal Usage:

- Write clear, descriptive natural language instructions
- Use semantic element descriptions (e.g., "Submit button" not "button at position 3")
- Avoid overly specific locators in manual overrides
- Review healing logs to identify UI stability issues

Limitations:

- Works best with semantically meaningful element interactions
- Requires clear original natural language instructions
- Performance may vary with complex, dynamic UIs

Integration with HyperExecute

Auto-Heal is tightly integrated with HyperExecute execution:

- Activated automatically for all KaneAI-generated tests
- No additional configuration in HyperExecute required
- Healing results included in standard test reports
- Supports all browser and device combinations on HyperExecute

Data Management & Security

Data Types & Storage

Test Data Categories

1. **Test Cases**: Structured test definitions
2. **Execution Results**: Pass/fail status, logs, metrics
3. **Variables & Secrets**: Dynamic and sensitive data
4. **Artifacts**: Screenshots, videos, reports
5. **Configuration**: Settings, preferences, capabilities

Storage Architecture

- **Database**: Relational database for structured data
- **Object Storage**: S3-compatible storage for artifacts
- **Vault**: HashiCorp Vault for secrets
- **Cache**: Redis for session data

Security Measures

Encryption

- **AES-256 for data at rest**: All data encrypted in database
- **TLS for data transmission**: All API calls encrypted in transit
- **End-to-end encryption**: Secrets encrypted from client to vault

TLS

- Encrypted data transmission
- Certificate pinning
- HTTPS only
- TLS 1.2+ required

Vault Integration

- HashiCorp Vault for secrets management
- Secure secret storage
- Access control and audit
- Secret rotation support

Access Control

- Role-based permissions (RBAC)
- Organization-level access control
- Project-level access control
- Resource-level permissions

Compliance

- **SOC 2 certified**: Type II compliance
- **GDPR certified**: Data privacy compliance
- **ISO 27001**: Information security management
- **HIPAA**: Healthcare data protection (enterprise)

Data Privacy

User Data

- Personal data minimization
- Data retention policies
- Right to deletion
- Data export capabilities

Test Data

- Secure test data storage
- Data isolation per organization
- No cross-organization data access
- Data anonymization options

Audit & Monitoring

Audit Trails

- All user actions logged
- Secret access tracked
- Configuration changes recorded
- API access logged

Monitoring

- Real-time security monitoring
- Anomaly detection
- Access pattern analysis
- Compliance monitoring

Performance Optimization

Parallel Execution

- Concurrent test execution
- Configurable concurrency
- Resource optimization
- Load balancing

Caching

- Session caching
- Asset caching
- API response caching
- Performance optimization

Resource Management

- Dynamic resource allocation
- Auto-scaling
- Queue management
- Resource cleanup

Advanced Test Capabilities

Multi-Tab/Window Testing

- Test across browser contexts
- Handle pop-ups
- New window interactions
- Tab switching

iFrame Handling

- Automatic iFrame detection
- Context switching
- Nested iFrame support

Shadow DOM

- Shadow DOM element detection
- Shadow root traversal
- Web component testing

File Operations

- File upload
- File download
- File validation

- Multiple file formats

Advanced Assertions

- Visual assertions
- Network assertions
- Database assertions
- API assertions
- Custom assertions

Core Architecture & Infrastructure

KaneAI Architecture Components

1. Frontend Layer

User Interface

- Web application (React-based)
- Mobile-responsive design
- Real-time updates (WebSocket)

Natural Language Input Processing

- Text input processing
- Voice input support
- Slash command parser
- Intent recognition

Real-time Test Execution Viewer

- Live session monitoring
- Step-by-step execution display
- Screenshot/video streaming
- Network activity visualization

2. API Gateway

Request Routing

- TLS encrypted communication
- Load balancing across services
- Rate limiting
- Request validation

Authentication & Authorization

- JWT-based authentication
- OAuth 2.0 support
- API key management
- Session management

Load Balancing

- Distributed request handling
- Auto-scaling
- Health checks
- Failover support

3. Core Services

Auth Server

- User authentication
- Session management
- Token generation and validation
- SSO integration
- MFA support

Test Management Service

- Test case CRUD operations
- Test run management
- Project and folder operations
- Metadata management
- Version control

VM Management Service

- Virtual machine provisioning
- Device allocation
- Resource pool management
- Session lifecycle management
- Cleanup and recycling

LLM Engine (Azure OpenAI)

- Natural language processing
- Test generation
- Scenario analysis
- Intent extraction
- Code generation assistance

Multi-Turn LLM Architecture (Agent Authoring Pipeline)

The KaneAI Agent uses a sophisticated multi-turn LLM pipeline to process natural language instructions and convert them into executable test steps. This architecture differs between web and mobile platforms.

Overview: Step Processing Pipeline

- `┌──────────────────────────────────────────────────────────────────────────────┐`
- `│                    Natural Language Instruction                    │`
- `│                 (e.g., "Click on the Login button")                 │`
- `└──────────────────────────────────────────────────────────────────────────────┘`
- `↓`
- `┌──────────────────────────────────────────────────────────────────────────────┐`
- `│                       BIFURCATION PROMPT                       │`
- `│  Determines: Is this instruction DETERMINISTIC or GENERATIVE?  │`
- `│                                                                │`
- `│  Deterministic: Explicit action with clear target              │`

- │   (e.g., "Click on Login button", "Type 'hello' in username field")   │
- │
  │
- │   Generative: Requires AI exploration to figure out steps
  │
- │   (e.g., "Login to the application", "Complete the checkout flow")   │
- └──┘
- ↓   ↓
- ┌────────────────────┐
  ┌────────────────────┐
- │   DETERMINISTIC   │   │   GENERATIVE
  │
- │   FLOW   │   │   FLOW
  │
- └────────────────────┘
  └────────────────────┘
- ↓   ↓
- ┌──────────────────────────────────────┐
  ┌──────────────────────────────────────┐
- │   TAGIFICATION PROCESS   │   │   GENERATIVE PROMPT   │
- │   │   │
  │
- │ - Analyze current screen DOM/UI   │   │ Inputs:
  │
- │ - Identify all interactable   │   │ - Current screen (visual)   │
- │   elements   │   │ - Tagified screen (annotated)   │

- │ - Assign unique tags to elements   │   │ - Original instruction   │
- │ - Create element map with   │   │ - Page context │
- │   locators (XPath, CSS, ID)   │   │ │
- │ - Generate confidence scores   │   │ Process: │
- │   │   │ - AI analyzes screen state   │
- └─────────────────────────────────────┘   │ - Determines next atomic step   │
- ↓   │ - Breaks complex instruction into   │
- │   │   smaller deterministic actions   │
- │ └─────────────────────────────────────┘
- │ ↓
- │ ┌─────────────────────────────────────┐
- │   │   TAGIFICATION PROCESS   │
- │   │   (Same as deterministic flow)   │
- │ └─────────────────────────────────────┘
- ↓ ↓
- ┌──┐
- │ ACTOR PROMPT │

- │ │
- │ - Receives tagified screen + instruction │
- │ - Maps instruction to specific element(s) │
- │ - Generates Selenium/Appium action command │
- │ - Includes multiple locator strategies for resilience │
- │ - Executes the action on browser/device │
- └──┘
- ↓
- ┌──┐
- │ ACTION EXECUTION │
- │ │
- │ Web: Selenium WebDriver action │
- │ Mobile: Appium action │
- │ Result: Success / Failure │
- └──┘
- ↓
- (For Generative: Loop back to check if goal achieved)

Bifurcation Prompt

The bifurcation prompt is the entry point for all natural language instructions. It analyzes the instruction to determine the processing path:

Deterministic Classification Criteria:

- Instruction contains explicit action verb (click, type, scroll, hover, select)
- Target element is clearly specified
- No ambiguity about what to do
- Single atomic action

Generative Classification Criteria:

- Instruction describes a goal or outcome
- Multiple steps may be required
- AI needs to explore the UI to determine actions
- Workflow-level instructions

Examples:

Instruction	Classification	Reason
"Click on the Login button"	Deterministic	Clear action + target
"Type 'john@example.com' in email field"	Deterministic	Clear action + target + value
"Login to the application"	Generative	Goal-based, multiple steps needed
"Add a product to cart and checkout"	Generative	Complex workflow
"Scroll down"	Deterministic	Clear action
"Find and select the cheapest option"	Generative	Requires exploration

Tagification Process

Tagification is the process of analyzing the current screen and identifying all interactable elements:

Process Steps:

1. **Screen Capture**: Capture current DOM state (web) or UI hierarchy (mobile)
2. **Element Detection**: Identify all interactable elements (buttons, inputs, links, etc.)
3. **Tag Assignment**: Assign unique identifiers to each element
4. **Locator Generation**: Generate multiple locator strategies per element:
   - XPath (primary)
   - CSS Selector
   - ID (if available)
   - Text content
   - Accessibility labels (mobile)
5. **Confidence Scoring**: Assign confidence scores to each locator based on stability
6. **Element Map Creation**: Create structured map of all elements with their attributes

Output: Tagified screen representation that the Actor prompt can use to identify targets.

Generative Prompt

For generative instructions, the AI uses additional context to determine the next atomic step:

Inputs:

- Current screen visual (screenshot)
- Tagified screen (annotated element map)
- Original natural language instruction
- Conversation history (previous steps in this flow)
- Page/app context

Process:

1. Analyze current screen state
2. Compare with goal from original instruction
3. Determine the most logical next atomic action
4. Output: A deterministic instruction for the Actor prompt

Loop: For complex workflows, the generative prompt loops until the goal is achieved or the system determines the goal cannot be achieved.

Actor Prompt

The Actor prompt is responsible for executing the actual action:

Responsibilities:

1. Receive tagified screen + specific instruction
2. Map instruction to the correct element using tags
3. Generate the appropriate WebDriver/Appium command

4. Include fallback locators for resilience
5. Execute the action

Output: Selenium/Appium command with execution result

Platform-Specific Differences

Web Authoring (Desktop & Mobile Browser)

- Uses DOM-based tagification
- Selenium WebDriver for execution
- iFrame context switching handled automatically
- Shadow DOM traversal supported
- Multi-tab/window support

Mobile App Authoring (Native Apps)

- Uses UI hierarchy for element detection
- Appium for execution
- Accessibility IDs prioritized for stability
- Platform-specific element attributes (Android: resource-id, iOS: accessibility-id)
- **Qwen Model Fallback**: When element detection fails during tagification, the Qwen model provides coordinate-based click fallback (see next section)

Qwen Model for Mobile Fallback

For mobile app and mobile browser authoring, KaneAI uses the Qwen vision model as a fallback mechanism when the standard tagification process fails to identify elements:

Trigger Conditions:

- Element not found in standard tagification
- UI hierarchy parsing fails
- Non-standard UI components
- Custom rendered views

Qwen Fallback Process:

1. **Screen Capture**: Capture visual screenshot of current screen
2. **Visual Analysis**: Qwen model analyzes the screenshot visually
3. **Element Location**: Model identifies target element position based on visual appearance
4. **Coordinate Generation**: Returns X,Y coordinates for the target element
5. **Coordinate-Based Action**: Appium performs tap/click at specified coordinates

Use Cases:

- Canvas-based applications
- Game UIs
- Custom-rendered components without accessibility labels
- Hybrid apps with WebView rendering issues

Limitations:

- Coordinate-based clicks are less stable across device resolutions
- No element verification possible
- May require recalibration for different screen sizes

- ┌───┐
- │ Standard Tagification │
- │ FAILS │
- └───┘
- ↓
- ┌───┐
- │ Qwen Vision Model │
- │ │
- │ Input: Screenshot + Target element description │
- │ Process: Visual element identification │
- │ Output: X,Y coordinates for tap action │
- └───┘
- ↓
- ┌───┐

- │   Coordinate-Based Tap
  │
- │   via Appium TouchAction
  │
- └──
  ┘

4. Data Layer

Database (AES-256 Encrypted)

- Test case storage
- User data management
- Execution results
- Configuration data
- Relational database (PostgreSQL)

Storage

- Test artifacts (S3-compatible)
- Screenshots and videos
- Execution logs
- Generated code
- Binary files (APK/IPA)

5. Execution Infrastructure

HyperExecute

- Scalable test execution engine
- Distributed test orchestration
- Parallel execution management
- Resource optimization
- Result aggregation

Device Cloud

- Real devices for testing
- Browser environments
- Mobile devices (iOS/Android)

- Desktop browsers
- Emulators and simulators

Network Services

- Tunnel service for local testing
- Proxy service
- Geolocation service
- Network throttling
- SSL/TLS handling

Test Case Generator Architecture

The Test Case Generator operates as a separate service within the ecosystem:

Input Processing

- **Text**: Natural language processing
- **Media**: Image and video analysis
- **Voice**: Speech-to-text conversion
- **Documents**: PDF and Word parsing
- **Third-party**: Jira/ADO API integration

LLM Processing

- Azure OpenAI integration
- Prompt engineering
- Context enrichment
- Validation layer for AI-generated content
- Quality assurance checks

Output Generation

- Structured test cases with steps and assertions
- Test scenario categorization
- KaneAI Test Plan creation
- Metadata extraction

System Architecture Diagram

- ┌──┐

```
│                    Frontend Layer                       │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐   │
│  │  Web UI      │  │ Mobile UI    │  │ Real-time    │   │
│  │              │  │              │  │   Viewer     │   │
│  └──────────────┘  └──────────────┘  └──────────────┘   │
└─────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────┐
│                    API Gateway                          │
│         (Authentication, Rate Limiting, TLS)            │
└─────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────┐
│                    Core Services                        │
│  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐    │
│  │  Auth    │ │   Test   │ │   VM     │ │  LLM     │    │
│  │ Server   │ │Manager   │ │Manager   │ │ Engine   │    │
│  └──────────┘ └──────────┘ └──────────┘ └──────────┘    │
└─────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────┐
│                      Data Layer                         │
│  ┌──────────────────────┐  ┌──────────────────────┐     │
│  │ Database (AES-256)   │  │ Object Storage       │     │
│  │ (PostgreSQL)         │  │ (S3-compatible)      │     │
│  └──────────────────────┘  └──────────────────────┘     │
│  ┌──────────────────────┐                               │
│  │  HashiCorp Vault     │                               │
│  │  (Secrets)           │                               │
│  └──────────────────────┘                               │
└─────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────┐
│              Execution Infrastructure                   │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐   │
│  │ HyperExecute │  │ Device Cloud │  │   Network    │   │
│  │              │  │              │  │   Services   │   │
│  └──────────────┘  └──────────────┘  └──────────────┘   │
└─────────────────────────────────────────────────────────┘
```

Scalability & Performance

Horizontal Scaling

- Microservices architecture
- Independent service scaling
- Load balancing
- Database read replicas

Auto-scaling

- Dynamic resource allocation
- Peak load handling
- Cost optimization
- Queue management

Caching Strategy

- Redis for session data
- CDN for static assets
- API response caching
- Database query caching

High Availability

- Multi-region deployment
- Failover mechanisms
- Data replication
- Disaster recovery

API & Programmatic Access

Core APIs

Test Runs Execution API

Endpoint

- `POST https://test-manager-api.lambdatest.com/api/atm/v1/hyperexecute`

Authentication

- `Authorization: Basic <base64(username:access_key)>`
- `Content-Type: application/json`

Key Parameters

Parameter	Type	Required	Description

`test_run_id`	string	Yes	Unique test run identifier
`concurrency`	integer	No	Parallel execution count (default: 5)
`environment`	string	No	Target environment name
`replaced_url`	string	No	Dynamic URL mapping for application under test
`region`	string	No	Execution region (US, EU, APAC)
`android_app_id`	string	No	Android app override for mobile tests
`ios_app_id`	string	No	iOS app override for mobile tests
`network_throttle`	string	No	Network profile (2G, 3G, 4G)
`tunnel`	string	No	Tunnel name for local testing
`retry_on_failure`	boolean	No	Enable automatic retry (default: false)
`max_retries`	integer	No	Maximum retry attempts (default: 0)

Request Example

- `{`
- `  "test_run_id": "TR-12345",`
- `  "concurrency": 10,`
- `  "environment": "staging",`
- `  "replaced_url": "https://staging.example.com",`
- `  "region": "US",`
- `  "network_throttle": "3G",`
- `  "tunnel": "my-tunnel",`
- `  "retry_on_failure": true,`

- `  "max_retries": 2`
- `}`

Response Structure

- `{`
- `  "job_id": "550e8400-e29b-41d4-a716-446655440000",`
- `  "app_job_id": "660e8400-e29b-41d4-a716-446655440001",`
- `  "test_run_id": "TR-12345",`
- `  "job_link": "https://hyperexecute.lambdatest.com/job/550e8400",`
- `  "mobile_job_link": "https://mobile-automation.lambdatest.com/build/660e8400"`
- `}`

Response Fields

Field	Type	Description
`job_id`	string (UUID)	HyperExecute job identifier
`app_job_id`	string (UUID)	Mobile job identifier (if mobile tests included)
`test_run_id`	string	Original test run ID from request
`job_link`	string (URL)	Link to HyperExecute job dashboard
`mobile_job_link`	string (URL)	Link to mobile automation dashboard

Status Codes

Status Code	Description
200	Success - Test run execution initiated

400	Bad Request - Invalid parameters
401	Unauthorized - Invalid credentials
404	Not Found - Test run not found
429	Too Many Requests - Rate limit exceeded
500	Internal Server Error - System error

Test Case API

Get Test Case

- `GET /api/v1/test-cases/{test_case_id}`

Create Test Case

- `POST /api/v1/test-cases`

Update Test Case

- `PUT /api/v1/test-cases/{test_case_id}`

Delete Test Case

- `DELETE /api/v1/test-cases/{test_case_id}`

Test Run API

Get Test Run

- `GET /api/v1/test-runs/{test_run_id}`

Create Test Run

- `POST /api/v1/test-runs`

Get Test Run Results

- `GET /api/v1/test-runs/{test_run_id}/results`

Project & Folder API

List Projects

- `GET /api/v1/projects`

List Folders

GET /api/v1/projects/{project_id}/folders

Variable & Secret API

Create Variable

- `POST /api/v1/variables`

Create Secret

- `POST /api/v1/secrets`

Rate Limits

API Rate Limits

- **Free tier**: 100 requests/hour
- **Paid tier**: 1000 requests/hour
- **Enterprise**: Custom limits

Concurrency Limits

- Based on subscription plan
- Configurable per test run
- Queue management for excess requests

Platform Information

Platform Limitations & Considerations

KaneAI Free Trial Limitations

Session Limits

- **2 AI Agent Sessions**: Maximum 2 parallel sessions
- **2 Test Manager Seats**: Limited to 2 users
- **10 sessions max**: Total session limit
- **40 instructions per session**: Maximum instructions allowed
- **10 minutes per session**: Session duration limit
- **2-minute idle timeout**: Session terminates after 2 minutes of inactivity

Testing Constraints

- **Single-tab testing only**: Multi-tab support not available
- **Limited device access**: Freemium devices only
- **Framework restrictions**:
  - Web: Selenium-Python only
  - Mobile: Appium-Python only

Feature Restrictions

- **Maximum 5 app uploads**: Limited mobile app storage
- **Network throttling**: Paid feature only
- **Smart variables**: Paid feature only

- **Parameter support**: Paid feature only
- **TOTP authentication**: Paid feature only
- **GPS location**: Paid feature only
- **Advanced mobile features**: Limited availability

Technical Constraints

Session Management

- Session timeout based on tier:
    - Free: 2 minutes idle timeout
    - Starter: 5 minutes idle timeout
    - Professional: 10 minutes idle timeout
- Maximum concurrent sessions per organization based on subscription
- Queue management when concurrency limit reached

Code Generation

- Only available for KaneAI Authored test cases
- Requires all steps to be successfully authored
- No skipped or failed steps allowed
- Framework availability varies by subscription

Execution Limits

- Concurrency based on subscription plan
- Rate limits on API calls
- Storage limits for artifacts
- Retention period for execution data

Browser & Device Support

Supported Browsers (Web)

- Chrome (latest and previous versions)
- Firefox (latest and previous versions)
- Safari (latest version)
- Edge (latest version)
- Opera (limited support)

Supported Mobile Platforms

- Android: 5.0 and above

- iOS: 11.0 and above
- Real devices and simulators/emulators

Known Limitations

Test Authoring

- Module nesting not supported (modules cannot reference other modules)
- Maximum step timeout: 300 seconds (5 minutes)
- Maximum test steps per test case: 100 steps (recommended)

Mobile Testing

- Geolocation mocking: Mobile devices only
- Biometric testing: Real devices only
- Camera injection: Limited device support

Integration

- Third-party API rate limits apply
- Sync frequency limitations
- Webhook retry limits

Future Roadmap & Beta Features

Currently in Beta

GitHub App Integration

- Pull request testing
- Code change analysis
- AI-powered test generation from PRs
- **Status**: Open beta
- **Availability**: All paid plans

Advanced Mobile Features

- Enhanced biometric testing
- Advanced gesture support
- Performance profiling
- **Status**: Limited beta
- **Availability**: Enterprise plans

Framework Support

- **Playwright** (JavaScript, Python): Beta
- **Cypress** (JavaScript): Beta
- **WebdriverIO** (JavaScript): Beta
- **Status**: Beta testing
- **Availability**: Selected users

Feature Requests

- Submit feature requests via channels and create LTPM Jira tickets for tracking
- Community voting on feature requests
- Regular roadmap updates
- Quarterly feature releases

Documentation & Resources

Official Documentation

- **Main Docs**: https://www.lambdatest.com/support/docs/
- **API Reference**: https://www.lambdatest.com/support/api-doc/
- **KaneAI Docs**: https://www.lambdatest.com/support/docs/kaneai/
- **Test Manager Docs**: https://www.lambdatest.com/support/docs/test-manager/

Access Points

KaneAI Platform

- **Agent**: https://kaneai.lambdatest.com/agent
- **Variables**: https://kaneai.lambdatest.com/variables
- **Modules**: https://kaneai.lambdatest.com/module
- **Sessions**: https://kaneai.lambdatest.com/sessions

Test Manager Platform

- **Projects**: https://test-manager.lambdatest.com/projects
- **Test Cases**: https://test-manager.lambdatest.com/test-cases
- **Test Runs**: https://test-manager.lambdatest.com/test-runs
- **Reports**: https://test-manager.lambdatest.com/reports

Support Channels

Email Support

- **General Support**: support@lambdatest.com
- **Enterprise Support**: enterprise@lambdatest.com

Enterprise Support

- Dedicated account manager
- Priority support
- Custom SLA
- Onboarding and training
- Regular check-ins

Community Support

- Community forum

Learning Resources

Video Tutorials

- Getting started guides
- Feature deep-dives
- Best practices

Webinars

- Monthly product webinars
- Q&A sessions

Blog & Articles

- Product updates
- Testing best practices
- Industry trends
- Case studies

Certification Program

- KaneAI Certification (coming soon)
- Test Automation Certification
- Advanced Testing Certification

Release Notes

Version Updates

- Monthly feature releases
- Weekly bug fixes
- Security patches

Change Log

- Detailed change documentation
- Breaking changes highlighted
- Migration guides
- Deprecation notices

Pricing & Plans

Plan Comparison

Free Trial

- 2 parallel sessions
- 2 Test Manager seats
- 10 sessions total
- Limited features
- Community support

Starter Plan

- 5 parallel sessions
- 5 Test Manager seats
- Standard features
- Email support
- Basic integrations

Professional Plan

- 25 parallel sessions
- 25 Test Manager seats
- Advanced features
- Priority support
- All integrations

Enterprise Plan

- Custom parallel sessions
- Unlimited Test Manager seats
- All features
- Enterprise support
- Custom integrations
- SLA guarantees
- Dedicated account manager

Add-ons

- Additional parallel sessions
- Additional Test Manager seats
- Extended storage
- Custom retention
- Advanced security features

●