# AI Browser Extension 1.0

Paul Hanchett

## Introduction

The AI Browser Extension (AIBE) is an extension for Chrome based browsers. It provides two sets of functionality:

1. To enable an AI to act as an observer, watching human interactions with web pages, potentially recording the interactions to be used for training,
2. And, to enable an AI to act on the web page, via the browser, just as a human would.

It's different from other browser control tools because it has a different perspective: Rather than focusing on *controlling* the browser (which it can do!), it focuses on watching what a human user is doing in the browser so it can learn to imitate the human.

Key to these activities is that the observer and actor channels are constrained to seeing only what a human viewing the browser page could see, and to doing only the actions a human user could do. By limiting actions and viewability in this fashion, the browser becomes simpler to control (keystrokes and mouse actions only), and the screen is simpler to view and interpret (only text and controls that are visible on the screen are reported, in the order they appear on the screen, with labels identifying targets for clicks and keystrokes).

Currently the control endpoint for the browser is a local web server, with inbound (observer) events from the browser viewable on the server and optionally recorded in a MongoDB database. The Actor control channel (events back to the browser) is also complete.

## How do we think this Extension will be used?

1. The first use of this extension will be to record human browser sessions for analysis and evaluation. Possibly, to eventually become AI training material.
2. The Observer capability could also be used to teach a nascent AI the basics of browser navigation.
3. Eventually, the Actor capacity can be used to close the loop, providing success feedback to the learning AI.
4. The AI could be trained to read text on the screen and surmise the screen's purpose, projecting what's needed to complete the screen, or what screen to seek to accomplish some purpose.
5. With enough exposure, an AI might be able to predict 'I need to find these kinds of websites, to complete my quest.'

The above is very hand-wavey because it posits how an entity might discover how to do a job. The above is pretty linear and AI's famously are not, but the extension will give AI a capability that is not available today.

## Installation

The AI Browser Extension consists of two components that work together:

1. **AIBE Server** - Python server (installed via PyPI)
2. **Chrome Extension** - Browser extension (downloaded from server)

### Installation from PyPI (Recommended)

**System Requirements:**

- Python 3.8 or higher
- Google Chrome browser
- Internet connection for PyPI download

**Step 1: Install AIBE Server**

Create a virtual environment (recommended to avoid conflicts):

```
# Create virtual environment
python -m venv .venv

# Activate virtual environment
# Windows:
.venv\Scripts\activate
# Linux/macOS/WSL:
source .venv/bin/activate

# Install from PyPI
pip install aibe-server
```

**Step 2: Install Chrome Extension**

Start the server and find the extension installation location:

```
# Start the server
aibe-server start

# Check where extension files are located
aibe-server status
```

**Example output:**

```
AI Browser Extension Server Status
========================================
+ Server running on Windows
   PID: 22368
   URL: http://localhost:3001
   Mode: NORMAL
   Uptime: 271047.9 seconds
   Started: 2025-11-26T15:05:46.036650
   Extension files: C:\Users\paulh\PycharmProjects\ai-browser-
extension\server\aibe_server\extension
   Active sessions: 1
   Events: 0
   Pending commands: 0

Commands:
```

```
  python server_manager.py stop     - Stop server
  python server_manager.py restart  - Restart server
```

Continue with extension installation:

```
# Visit the installation page
# Open Chrome and go to: http://localhost:3001/extension/install
```

Follow the on-screen instructions to download and install the Chrome extension in Developer Mode. The status command shows you exactly where the server is installed, which helps locate the extension files.

**Step 3: Install MongoDB (Recommended)**

Refer to the *Installation* section of the MongoDB Appendix. Installation of MongoDB is not *required* to use this extension, but it will make accessing and analyzing events on the observer stream much more convenient.

## Platform-Specific Notes

**Windows:**

- Use Command Prompt or PowerShell
- Virtual environment activation: `.venv\Scripts\activate`
- Server runs as background process when started

**Linux/macOS:**

- Use Terminal
- Virtual environment activation: `source .venv/bin/activate`
- Server runs as detached background process

**WSL (Windows Subsystem for Linux):**

- Install in WSL environment using Linux commands
- Server localhost:3001 accessible from both WSL and Windows due to network mirroring

- Can manage server from either Windows or WSL terminals
- Chrome extension installation works from Windows Chrome browser

## Development Installation

For developers wanting to modify the source code:

> ***Note:*** *Source repository access requires permission. Contact paul@paulhanchett.com for contributor access.*

```
# Clone repository (requires access)
git clone <repository-url>
cd ai-browser-extension

# Install server in development mode
cd server
python -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install -e .

# Load Chrome extension from local files
# Load unpacked extension from: server/extension/
```

## Building the Extension

**Windows Convenience Script:**
For Windows users, a convenience script `rebuild-distribution.bat` is available in the project root. This script cleans previous artifacts and builds the Python distribution packages. Note that you may still need to build the Chrome extension JavaScript separately as described below.

The Chrome extension source code is modularized (ES6 modules) but must be bundled for use in Chrome (Manifest V3 content scripts don't support ES6 modules directly).

**Prerequisites:**

- Node.js and npm installed

**Build Process:**

1. Navigate to the extension directory:

```
cd server/aibe_server/extension
```

2. Install dependencies (esbuild):

```
npm install
```

3. Build the extension:

```
npx esbuild content.entry.js --bundle --outfile=content.js --format=iife
```

**Source Structure:**

- `modules/` - Source code (ES6 modules)
- `content.entry.js` - Entry point importing all modules
- `content.js` - Generated bundle (do not edit directly)

**Build Verification:**

Before packaging for PyPI distribution, verify that `content.js` is up to date with source files:

```
# From the server directory
python verify_build.py
```

This checks that `content.js` is newer than all source files in `modules/` and `content.entry.js`. The script will fail with a clear error message if a rebuild is needed, preventing stale code from being packaged.

## Server Management

The AIBE server provides a complete command-line interface for managing the background server process.

## Basic Commands

```
aibe-server start     # Start server in background
aibe-server status    # Check if server is running
aibe-server stop      # Stop the server
aibe-server restart   # Stop then start
aibe-server help      # Show command help
```

## Command Details

**Start Server:**

```
aibe-server start
```

- Starts server on `http://localhost:3001`
- Runs in background (detached process)
- Shows server PID and installation path
- Detects if server already running

**Check Status:**

```
aibe-server status
```

- Shows server running status
- Displays installation path, PID, uptime
- Shows active sessions and event counts
- Indicates platform (Windows/Linux/WSL)

**Stop Server:**

```
aibe-server stop
```

- Gracefully shuts down server
- Works across platforms (Windows/Linux/WSL)
- Handles cross-platform process management

**Restart Server:**

```
aibe-server restart
```

- Stops existing server, then starts new instance
- Useful during development or configuration changes

**Help:**

```
aibe-server --help
# or
aibe-server help
```

- Shows complete command reference
- Includes usage examples

## Cross-Platform Behavior

**Same Machine, Different Environments:**

- Server can run in Windows, WSL, or Linux
- Commands detect which environment server is running in
- Provides appropriate management instructions
- WSL network mirroring makes localhost:3001 accessible from both Windows and WSL

**Process Isolation:**

- Each platform maintains separate server processes
- Windows and WSL can each run their own server instance
- Status command indicates which platform server is running on
- Stop/restart commands only affect servers in the same environment

## Configuration

**Default Settings:**

- Server Port: 3001
- Server Host: localhost (127.0.0.1)
- Log Level: INFO
- Background Mode: Enabled by default

**Environment Variables:**

```
AIBE_PORT=3001          # Change server port
AIBE_DEBUG=true         # Enable debug logging
AIBE_LOG_LEVEL=DEBUG    # Set logging verbosity
```

**Web Configuration Interface:**

AIBE provides a web-based configuration editor accessible at:

```
http://localhost:3001/config
```

**Default credentials:**

- Username: `admin`
- Password: `admin123`

The web interface allows you to:

- View current server configuration
- Edit MongoDB connection settings
- Configure test and server streaming options
- Modify authentication settings

**Configuration file locations:**

- Active config: `~/.AIBE/config.json` (created automatically on first run)
- Sample config: `server/aibe_server/AIBE_config_sample.json` (reference

template)

**When to use:**

- **Web UI**: Quick configuration changes, visual overview of settings
- **Direct file edit**: Bulk changes, configuration templates, version control

For detailed configuration options, see the MongoDB Appendix section "MongoDB Configuration."

### Verification

After installation, verify everything works:

1. **Server Status:**

   ```
   aibe-server status
   ```

   Should show "Server running" with PID and uptime.
2. **Web Interface:**
   Visit `http://localhost:3001/status` in browser - should display server information.
3. **Extension Status:**
   Click Chrome extension icon - should show "Connected" status.
4. **Event Capture:**
   Browse any website with extension active - events should be captured automatically.

## Sessions

A user may have more than one tab open in their browser, and it would be confusing to mix them all together. So the server backend assigns a sessionID to each tab the user opens, and uses that to identify the source tab of each event.

### Session Management Endpoints

The endpoints below are used by the extension injected script, content.js, to manage sessions. Use of these endpoints by the extension is automatic:

- `PUT /sessions/init` - Initialize new session with TabIdentity data
- `GET /sessions` - List all active sessions (returns array of SessionInfo objects)
- `POST /sessions/close` - Close session when tab is closed
- `POST /sessions/{session_id}/heartbeat` - Update session activity timestamp

## Observer Channel

### Load

Load is a navigation event, reporting that the browser is navigating to a new URL.


```
{
    "type": "event",
    "event": "load",
    "url": "http://localhost:3001/test-dropdown",
    "timestamp": "2025-07-03T18:23:25.395Z"
},
```


### Screen Status

Every event (mouse click or key press) *may* cause the screen to change. Screens are not reported unless they change. Once they change, they won't be reported until they stop changing. Changes in the screen content are reported as **screen_status** events.

The entire screen is scanned for visible textual content and for visual elements (like input or anchor elements), and then sorted by position top to bottom, left to right, putting all elements in their X, Y screen relationship to one another. If an control has a label assigned, the label is added to the control. If not, an algorithm attempts to locate the visual element a human would associate with the control on the screen.

Screen status reports the current URL of the page, which control has focus, the list of visible elements including text and controls, and the time that the screen stabilized as JSON. To reduce noise in the Observer stream, positional details and internal DOM references are omitted before events reach the server:


```
{
  "type": "screen_status",
  "url": "http://localhost:3001/test-dropdown",
  "focus_label": null,
  "focus_id": null,
  "visible_elements": [
    {
      "label": "Country",
      "control_type": "INPUT_DROPDOWN",
      "type": "select-one",
      "current_selection": "Select a country",
      "current_value": "",
      "clickable_options": [
        "\"United States\" → us",
        "\"Canada\" → ca",
        "\"United Kingdom\" → uk",
        "\"France\" → fr"
      ],
      "is_open": false,
      "is_disabled": false,
      "href": null
    },
    {
      "label": "Skills (Multi-select)",
      "control_type": "INPUT_DROPDOWN",
      "type": "select-multiple",
      "current_selection": "(none selected)",
      "current_value": null,
      "clickable_options": [
        "\"JavaScript\" → js",
        "\"Python\" → py",
        "\"Java\" → java",
        "\"CSS\" → css",
        "\"HTML\" → html",
        "\"React\" → react"
```

```
      ],
      "is_open": false,
      "is_disabled": false,
      "href": null
    }
  ],
  "timestamp": "2025-07-03T18:23:25.698Z"
}
```


Notice that the controls reported here are *not* HTML control names! To simplify the many variations on controls like **INPUT**, controls are named according to their semantic type…

On an HTML page, the INPUT control is a basic element, yet this one control maps onto many different behaviors that are mutually exclusive. Rather than carry that complexity into our design, we've created new input control types, that explicitly express the behaviors of each input variation:

**Semantic Control Types:**

| CONTROL TYPE | DESCRIPTION | HTML ELEMENTS |
|---|---|---|
| INPUT_TEXT | Single-line text input | `<input type="text">`, `<input type="email">`, `<input type="password">` |
| INPUT_TEXTAREA | Multi-line text input | `<textarea>` |
| INPUT_CHECKBOX | Checkbox control | `<input type="checkbox">` |
| INPUT_RADIO | Radio button | `<input type="radio">` |
| INPUT_DROPDOWN | Dropdown selection | `<select>` (single/multiple) |
| BUTTON | Clickable button | `<button>`, `<input type="submit">`, `<input type="button">` |
| LINK | Hyperlink | `<a href="...">` |
| INPUT_DATE | Date picker | `<input type="date">`, `<input type="datetime-local">` |
| INPUT_NUMBER | Numeric input | `<input type="number">`, `<input type="range">` |
| INPUT_FILE | File upload | `<input type="file">` |

These semantic types abstract away HTML implementation details, focusing on what the control does from a human perspective.

## Mouse and Keyboard events

Mouse clicks and keyboard presses are reported as events, and include the targets of those events. An event target will always be a visible (e.g., clickable) element on the screen.


```
{
  "type": "mouse",
  "event": "click",
  "target": {
    "label": "Country",
    "value": {
      "United States": "us"
    }
  },
  "button": 0,
  "buttons": 0,
  "timestamp": "2025-07-03T18:23:30.037Z"
}
```


Mouse and keyboard events are always directed at a target element. Consequently, those events will always have a `target` entry containing **both** the target element's `label` (or possibly an auto-generated identifier if a label can't be identified) **and** a `value` object. For state-changing events, `value` describes the final state of that control (for example, the selected dropdown options, the final checkbox state, or the final text of a field). For elements without an obvious value (such as a plain button), `value` is simply an empty object `{}`.

Consequently, the `target` entry can be used to locate the element this click (or keypress) occurred on, as well as the final values set on the element as a result (for example, the text entered into a field, the options selected in a dropdown, or the destination URL of a clicked link).

### Keyboard Event Handling

**Final Values Only**: The browser does not offer complete fidelity reporting keystrokes. So it's not possible to completely understand what actions were taken on a control. As a result, we report only the control's final value.

Keyboard events are consolidated to reduce noise. Text input is captured as single events containing the final text value. Control keys (Enter, Space, Arrow keys, Tab) are sent immediately as navigation events.

**Keyboard Event Format**:

```
{
  "type": "keyboard",
  "event": "text_input",
  "target": {
    "label": "Email Address",
    "value": { "text": "user@example.com" }
  },
  "timestamp": "2025-01-08T10:15:35.500Z"
}
```

**When Keyboard Events Are Sent:**

- When focus leaves the input field (blur)
- When user clicks elsewhere on the page
- When form is submitted
- When page navigation occurs (load, unload events)

**Control Keys**: Navigation keys (Enter, Tab, Arrow keys) and Space (when not in text input) are sent immediately as `control_key` events since they trigger actions rather than text input.

## How All Controls Are Reported

AIBE simplifies event reporting by focusing on final control states rather than tracking every intermediate action:

**Text Input Controls** (INPUT_TEXT, INPUT_TEXTAREA):

- Individual keystrokes are **not** reported separately
- Only the final text value after user completes input
- Target always shows the complete, final content

**Selection Controls** (INPUT_DROPDOWN, INPUT_RADIO, INPUT_CHECKBOX):

- Reports the final selected state
- Includes both human-readable label and internal value
- Multiple-selection dropdowns show all selected options

**Mouse Events**:

- Click location (x, y coordinates)
- Target control with final state at time of click
- Modifier keys (Ctrl, Shift, Alt, Meta) if pressed

**Why This Approach?**

First, the browser does not allow us to see *every* keystroke! Some, it interprets for itself and won't pass them on to us. We can't reconstruct the actual actions taken on a control to pass them to a remote viewer.

Secondly, tracking every keystroke or intermediate selection creates massive event noise without adding meaningful information. The AI only needs to know "what did the user enter?" not "What were the precise keystrokes?" Using final values:

- Reduces event volume by 90%+
- Focuses on intent rather than mechanics
- Makes pattern recognition clearer
- Works identically for both Observer (recording) and Actor (playback)

**Example**: User types "johndoe@example.com" into email field:

- **Not reported**: 21 individual keystroke events, backspace corrections, etc.
- **Reported**: One keyboard event with `value: { "text": "johndoe@example.com" }`

### Observer Endpoints

Observer endpoints (require sessionId):

- `POST /sessions/{session_id}/events` - Submit new browser event to specific session
- `GET /sessions/{session_id}/events/recent?limit=50` - Get recent events for session
- `GET /sessions/{session_id}/events/consume` - Consume unprocessed events (FIFO queue)
- `GET /sessions/{session_id}/events/unprocessed` - View unprocessed events for session
- `GET /sessions/{session_id}/events/processed?limit=50` - View processed events for session

## Actor Channel

The Actor channel is now fully operational. It enables a remote AI system to control the browser by sending commands that are executed as if a human user performed them.

**Observer Events as Actor Commands:**

The Actor channel accepts any Observer event as a command - it's designed to play back what was 'heard' on the Observer channel. However, for clarity, understand that not all actor events require browser action:

- **Actionable events**: `load`, `mouse`, `keyboard` - These execute as browser commands
- **Ignored events**: `screen_status` and other observational events - Accepted without error, but don't trigger actions

This design allows you to record an Observer session and replay it through the Actor channel, with the system intelligently filtering what needs action versus what's just state reporting.

**Event Recording and Source Labeling:**

When Actor commands execute in the browser, the Observer channel records them just like user actions, but labels them with `source: "ai"` to distinguish them from human user actions (`source: "user"` or omitted).

## Actor Command Types

The Actor channel supports three types of commands:

1. **Navigation Commands** (`load`): Navigate to a specific URL
2. **Mouse Commands** (`click`): Click on elements identified by their label
3. **Keyboard Commands** (`keyboard`): Type text into form elements, also identified by their element label.

The Actor channel is the reciprocal path to the Observer channel. As such, it accepts any valid observer event without error (for example, screen_status events), but does not act on all of them.

## Actor API Endpoints

Actor endpoints (require sessionId):

- `POST /sessions/{session_id}/actor/send` - Queue command(s) for specific session
- `GET /sessions/{session_id}/actor/commands` - Poll pending commands for session (used by browser extension)
- `GET /sessions/{session_id}/actor/retrieved?limit=50` - View previously retrieved commands for session
- `GET /sessions/{session_id}/status` - Get session status including Actor command counts

Server status endpoints:

- `GET /status` - Server status including Actor command statistics

Commands are queued on the server and retrieved by the browser extension via polling every 100ms, creating a reliable command delivery system with session isolation.

### Important: Browser Tab Visibility Requirement

**Critical:** The Actor channel requires the browser tab to be visible and active for proper operation. Chrome significantly reduces processing time allocated to hidden or background tabs, which may cause Actor commands to appear to hang or fail to execute.

When a tab is hidden, minimized, or in the background, Chrome-based browsers throttle:

- JavaScript execution timing
- DOM updates and screen rendering
- Event processing and element detection

This browser-level performance optimization impacts the extension's ability to detect screen changes, locate elements, and execute commands with proper timing.

> ***Always ensure the target browser tab is visible and active when using Actor channel commands or running automated tests.***

## Interpretation: Imposing Understanding on Browser Events

The Observer Channel captures raw browser events - clicks, keystrokes, screen updates. How do we make sense of this stream of data? AIBE organizes these events into a hierarchical structure that reveals natural patterns in web browsing behavior.

Here's how we aggregate user actions into what we'll call a Story—the history and sum of all user and browser actions, in a single browser tab:

### Browser Tab Opens

When the browser opens initially, or when the user creates a new tab, that tab is assigned a session ID by the extension. The session ID is unique and will stay with that tab as long as it is open. If the tab is closed and then reopened, it will still have its original ID and content. If the tab is closed and a new tab opened, it will have its own unique ID. (In the first case, reopening is like an undo of the closing of the tab. In the second, the tab is actually closed and a new one created to replace it.)

Each browser tab contains the tale of one *Story*, all events resulting from the user's browsing in that tab.

## Events

The user performs actions, and causes Events. These are a fundamental building block, for our Story:

- Individual mouse clicks
- Keyboard input (final values)
- Page navigation (load events)
- Consequential screen state updates
- (Events in the Actor channel also cause Observer events, marked with `source: AI`)

Each event has:

- Timestamp (when it occurred)
- Type (mouse, keyboard, load, screen_status)
- Target (which element was affected)
- Source ("user" or "ai" - who initiated it)

When a user clicks on a text field (Element) and types in a new value, the display updates to reflect the new value. In modern UIs, other fields may update as well, leading to a screen update. If the field were a dropdown box, the user would select an item as the new value, and again the screen would update. Sometimes multiple actions contribute to the final value of a field, generally but not always completed by an update of the display screen.

Finally, clicking on a button or on a link might change not only the screen, but also the current URL.

## Words

For purposes of understanding, we'll call all the consecutive changes to a single field, plus any consequential screen or URL updates, a *Word-- a unit of meaning created by the actions of the user, without regard to time*. Shifting away from the current element to another element or to another page ends the current Word and begins a new one because a new element is now the object of actions.

The collection of Words 'executed' on one page are significant because they "describe" what the user has done on that page. We might not understand the semantic significance of those Words, but we do know they are *associated* with the function of the web page.

## Sentences

The collection of consecutive Words associated with a single display URL (a page) will be our *Sentence*. When a Word moves the display to a new URL, that's the beginning of a new Sentence. In practice, a form submission could be to a different URL which might then redirect us back to the original URL. The intermediate submission URL "doesn't count", because the Human would never have seen it!

The (unspoken) subject of a sentence is the URL it's associated with. We'll come back to this when we talk about imputing meaning to the user's actions.

## Paragraphs

Paragraphs are built of Sentences. Sentences begin with a URL change, and are unified by sharing a common Domain Name. When the Domain Name changes, a new Paragraph begins. The Sentences of a Paragraph describe the operations performed at a single web site.

## Story

The story of a tab is the sequential collection of all Paragraphs in a browser tab. A tab has only one Story, because the Story begins with the creation of the browser tab, and concludes with the closing of that tab.

What do we have so far?

A Story tells the journey from domain to domain in its Paragraphs, as the user works with their browser. The Paragraphs tell us what domains were visited. The Sentences of each Paragraph tell what pages of the domain were visited, and the Words of each Sentence tell what was done to the elements of individual pages.

We've now defined a syntax for user activities, without any reference to a particular task!

There's more:

- Sentences are built around a single page. To be useable by humans, the web page designers need to Title the page, probably as a `<H1>` entry!
- Paragraphs revolve around individual web sites; get the site's name and you likely know what it does, too!
- Looking at what the user did and what they entered, you might be able to reason backwards to "What were they trying to do?" This entails thinking about action sequences as analogs to programming functions--We'll discuss this again, below.

There's another point to make: An experienced user might open more than one tab (or browser instance) to do a job. One tab might be incoming orders, and another tab might be available stock. To understand this user's browsing, we'd need to notice actions across two or more (not one) tabs!

While we're not going to try to work with that level of complexity yet, we can think of a collection of related Stories as a *Legend* (like the Legends of Robin Hood, or of King Arthur.)

Finally, if the user is just browsing randomly all this neat organization won't get us much, but we should be able to tell that, because nothing will be connected.

This system has one significant liability: it consumes significant AI context. The data structures involved can be huge, and won't easily fit in the usual AI context.

---

## The Natural Linguistic Structure

**The Analysis above, reveals that user activity has a natural linguistic structure**, without even considering what the user is trying to accomplish or the data they are entering (the semantic content!)

This understanding helps an AI observer assign meaning to web actions without regard to semantics, and to form hypotheses about how they are joined together to accomplish tasks.

By looking at headings and titles on individual pages (Paragraphs), we can infer generally what the page is for, and narrow what the user could be doing on that page. Then we can look at the Words they're choosing, and make an even better assessment of their goals.

By reviewing the analyzed Paragraphs of the Story we see the arc from where they started to where they ended. If we can guess the problem they were trying to solve (from the final Sentence of the last Paragraph), we can infer how they developed a solution.

---

*The linguistic structure we've described opens up some interesting possibilities for future exploration. While not yet implemented, these implications are worth considering here while the concepts are fresh:*

## Function Model Observation

**Why web pages are like programming functions:**

By observing what users enter on a page (inputs) and what appears afterward (outputs), we can model each Sentence as a function signature:

```
Sentence("search product page", inputs: {product_name,
price_range})
  → outputs: {product_list, availability, prices}
```

**This functional abstraction reveals something profound**: Even without understanding *what* the user is searching for or *why* they want it, we can learn:

- What inputs this page requires (form fields that must be filled)
- What outputs this page produces (information displayed afterward)
- Under what conditions this "function" succeeds or fails

**Why this matters - composing solutions from observed patterns:**

Having function signatures for multiple Sentences (think subroutines), we can work backward from desired outputs:

1. **Goal**: User wants "product availability in Portland"
2. **Function composition**: Search for Sentences that produce "availability" outputs
3. **Input discovery**: Trace back to find what inputs those Sentences require
4. **Path planning**: Chain Sentences together: location selector → product search → availability checker

**Cross-site learning without semantic understanding:**

Once we know the universe of available Sentences (functions) across many websites, we can possibly synthesize solutions to problems we haven't seen before:

- **Pattern recognition**: "This page looks like a search function" (inputs: criteria, outputs: results list)
- **Transfer learning**: "I've seen this pattern on Amazon, eBay, and Target - it's a product search"
- **Hypothesis-driven exploration**: "If I need product prices, I should look for pages with price-range inputs"

*Functions, inputs and outputs become a toolkit that can be assembled to get desired outputs from available inputs.*

**The breakthrough**: All this works without being able to name precisely what the user is doing! We don't need to understand "product search" semantically - we just need to recognize the input/output pattern.

This is precisely what a skilled operator learns through experience with multiple websites - they develop an intuition for "this site has a function that takes X and produces Y," and they chain those functions together to accomplish goals.

## Future Research: Multi-Story Relationships

Real-world problem-solving often spans multiple browser tabs working together. For example, debugging code might involve:

- Tab 1 (Story): Stack Overflow - researching the error message
- Tab 2 (Story): IDE or editor - implementing the fix
- Tab 3 (Story): API documentation - verifying correct usage
- Tab 4 (Story): Local test page - confirming the fix works

**Research Question**: Can we detect when multiple Stories are related to the same problem-solving session?

**Challenge**: Just because two Stories share a common subject doesn't mean they're connected. Multiple Stories might each relate to "debugging authentication" without any actual connection except the topic. We lack user motivation data to determine this prospectively - we can't know *why* the user opened each tab or how they're coordinating across tabs. By looking at timestamps we *can* observe that while working in one tab the user switched to another, did some work and returned, or switched to still another tab. If that switching behavior repeats it might tell us something about the user's workflow...

**Discovery Path**: Analysis of actual transaction data might reveal patterns - tab-switching frequency, shared content between tabs, or temporal clustering - that indicate coordinated work. This can only be known retrospectively, after examining real browsing sessions and looking for paths through the transactions.

**Noise Problem**: Current Stories may contain aimless browsing mixed with focused work. We don't yet know how to distinguish intentional problem-solving from random exploration. This is why formalization is premature - we need the "Action Super Collider" of real data and experiments to prove hypotheses about multi-tab coordination patterns.

---

*Back to our previous topic: the events from the Observer channel and the Stories they create are stored in MongoDB:*

## MongoDB Storage

Stories are stored in MongoDB with hierarchical structure matching the linguistic model:

```
Story {
  story_id: "tab_abc123...",
  start_time: "2025-01-08T10:00:00.000Z",
  paragraphs: [
    {
      domain: "github.com",
      sentences: [
        {
          url: "https://github.com/user/repo",
          words: [
            {
              target_label: "Search",
              events: [click, type, type, ...],
              screen_status: {...}
            }
```

```
          ]
        }
      ]
    }
  ]
}
```

## Design Rationale: Technical Deep Dive

The narrative above provides a conceptual understanding of the Story structure. This section explains the technical design decisions and "why not" alternatives considered during development.

### Why Element-Based Words?

#### WORD (Element Interaction)

All events targeting the same control form one **Word**.

**Why element-based, not time-based?**

Early designs considered time-based Words: "all events within 2 seconds form one Word." This failed because:

- **Typing speed variation**: Fast typists would create different Word "spellings" than slow typists for the same logical action
- **Network lag interference**: Screen updates happen unpredictably, arbitrarily splitting actions
- **No mental model match**: Users think "I filled in the username field" (one logical thought), not "I typed 8 characters in 1.5 seconds"

**Element-based boundaries solve this**: All events directed at the same control form one Word, regardless of timing. This matches human cognition - one target element equals one interaction unit.

**Example**: Typing "username" into login field = one Word (all keystrokes target same input control)

**Why Status Updates are separate from events:**

Status updates serve validation, not action recording:

- **Events capture actions**: "what the user did" (clicked, typed, selected)
- **screen_status captures state**: "what the screen showed afterward"

If we stored screen_status in the events array, it would appear to be a "user action" when it's actually "system observation." Separating them makes the distinction clear:

```
Word.events = [click, type, click]  // User performed these actions
Word.screen_status = {...}          // Screen showed this result
```

This separation enables AI to learn function signatures: "When I perform these actions, this screen state should result."

**Validation insurance**: Typing an email address creates one Word. The status update afterward confirms not only the email field value, but reveals if AJAX validation changed other fields simultaneously - error messages appearing, dependent dropdowns updating, submit buttons enabling.

## Why URL-Based Sentences?

### SENTENCE (Page Context)

All Words within one URL context form a **Sentence**.

**Boundary markers**: URL changes, load events

**Why observation-based, not prediction-based?**

A naive approach would try to predict Sentence boundaries: "Form submission probably means new page, so start new Sentence." This fails in real-world web applications.

**The ambiguity problem**:

- **Load events** clearly indicate new Sentences (navigation, refresh, back/forward buttons)

- **Form submissions** are ambiguous - impossible to predict behavior without executing them:
    - Some POST forms redirect to new URLs (confirmation pages) → new Sentence
    - Some reload the same URL (validation errors) → same Sentence? new Sentence?
    - Some update via AJAX without any navigation → definitely same Sentence
    - Some single-page apps change URLs without traditional loads → URL changed, but was it a "navigation"?

**Observation-based solution**: The system observes what actually happens (URL changes, load events) rather than trying to predict form submission or AJAX behavior. Let the browser tell us what occurred.

**Why this matters for AI training**: Predictive boundary detection would create inconsistent training data. The same form might be "new Sentence" or "same Sentence" depending on validation state. Observation-based boundaries capture authentic browser interaction patterns as they actually occur, giving AI consistent, reliable structural markers.

**Example**: Filling out login form (username Word + password Word + click submit Word) = one Sentence

## Why Domain-Based Paragraphs?

### PARAGRAPH (Site Workflow)

Sequence of Sentences within one domain forms a **Paragraph**.

**Boundary markers**: Domain changes

**Why domain boundaries matter**: Different sites = different contexts and purposes. GitHub has different workflows than Amazon. Insurance sites have different patterns than banking sites.

**Example**: Search repo → View file → Edit file → Commit = one Paragraph (GitHub workflow)

## Why Tab-Based Stories?

### STORY (Complete Tab Session)

All Paragraphs from tab open to tab close form a **Story**.

**Important**: Stories capture actual tab usage, not intentional task completion. A Story might contain focused work, random browsing, tangents, or any combination - this reflects real human web behavior patterns.

**Example**: Tab opened to debug error → Stack Overflow → Back to IDE → Test page → Close tab = one Story

# Testing Infrastructure

The AI Browser Extension includes a unified testing infrastructure that validates both Observer and Actor channels. The test suite covers 12 test suites with 100% pass rate across all web control types.

## Architecture

Both the console and web test runners share a common JavaScript framework served by the AIBE server:

| COMPONENT | SERVED VIA | PURPOSE |
|---|---|---|
| `TestingFramework.js` | `/TestingFramework.js` | Session management, navigation, element interaction, Observer stream processing |
| `GenericElementTest.js` | `/GenericElementTest.js` | Data-driven element testing with support for all control types |
| `DataDrivenTestRunner.js` | `/DataDrivenTestRunner.js` | Test suite execution engine with progress callbacks |

The framework files are located in `server/aibe_server/tests/framework/` and served by the Python server to both test interfaces.

> ***Maintainer Note:*** *When modifying the testing framework, always verify changes work in both console and web environments. Since the framework is served by the server, you must restart the server (`aibe-server restart`) for changes to take effect.*

## Running Tests

**Prerequisites (both interfaces):**

1. AIBE server running (`aibe-server start`)
2. A Chromium-based browser (Chrome, Edge, etc.) open to any `localhost:3001` page
3. Browser tab must be **visible and active** (Chromium throttles background tabs)

**Console Interface:**

```
cd server/aibe_server/tests
node console-runner.js [--log-level=LEVEL]
```

Options:

- `--log-level=ERROR` - Only errors and failures
- `--log-level=WARN` - Warnings and above
- `--log-level=INFO` - Normal operations (default)
- `--log-level=DEBUG` - Verbose debugging

You will see browser activity as the tests proceed. Let them run to completion (a couple of minutes).

**Web Interface:** Navigate to `http://localhost:3001/test-runner`

- Real-time progress tracking with visual feedback
- Interactive log level selection (persisted in localStorage)
- Expandable test suite details

## Test Coverage

The test suites exercise all semantic control types:

| SUITE | CONTROLS TESTED |
|---|---|
| `basic_inputs` | Text fields, email, password |
| `form_states` | Field updates, clearing values |
| `edge_cases` | Whitespace, special characters, missing elements |
| `dropdown_controls` | Single-select, multi-select dropdowns |
| `checkbox_controls` | Checkbox toggle states |
| `radio_controls` | Radio button groups |
| `textarea_controls` | Multi-line text input |
| `input_variations` | URL, search, and other input types |
| `password_controls` | Password masking behavior |
| `button_controls` | Button clicks, ARIA buttons |
| `link_controls` | Internal, external, mailto, tel links |
| `nested_form_controls` | Fieldset-nested controls |

## Server Endpoints

**Test Pages:**

- `GET /test-inputs` - Basic input fields
- `GET /test-controls` - Comprehensive controls (dropdowns, checkboxes, buttons, etc.)
- `GET /test-result?action={action}` - Result confirmation page

**Test Runner:**

- `GET /test-runner` - Web-based test runner interface

**Framework Libraries (served to both interfaces):**

- `GET /TestingFramework.js`

- `GET /GenericElementTest.js`
- `GET /DataDrivenTestRunner.js`

**Debugging:**

- `GET /sessions/explorer` - Interactive session examination

## Troubleshooting: Stale Sessions

Tests look for the first session matching `localhost:3001`. If you see test timeouts or failures, stale sessions may be the cause.

**The problem:** When a browser tab closes, the extension should notify the server—but closing the tab also removes the close hooks before they can fire. A heartbeat mechanism eventually detects closed sessions, but this takes time. Meanwhile, tests may try to use a stale session that no longer exists.

**Symptoms:**

- Tests time out waiting for screen updates
- Session explorer shows multiple `localhost:3001` sessions
- Tests worked before but now fail without code changes

**Solution:** Close the browser with the extension installed, then restart the AIBE server:

```
aibe-server restart
```

If you restart the server without closing the browser first, the browser may reinstate the stale session via heartbeat.

**Recommended practice:** Use two browsers—one with the extension installed (the Browser Under Test) and another without the extension for documentation, viewing test results, etc. This avoids accidentally creating extra sessions.

# Appendix -- MongoDB Integration

MongoDB is an optional component that provides persistent storage for Story Assembly and debugging streams. AIBE operates normally without MongoDB, but certain features require database connectivity.

## MongoDB Adds functionality, but is not required

**AIBE continues to work normally when MongoDB is unavailable.** The server gracefully degrades functionality:

✅ **When available:**

- Observer channel event capture
- Actor channel command execution
- Session management
- Testing infrastructure (48 automated tests)
- Server status endpoints
- Event queues and real-time processing

❌ **Unavailable:**

- Story Assembly persistence (Stories remain in memory only)
- ServerStreamer debugging (server event logging disabled)
- Long-term event history and pattern analysis

**Error Handling:**

When MongoDB is unavailable, AIBE logs errors but continues processing:

- Connection failures are caught and logged
- ServerStreamer silently disables itself
- Story Assembly attempts database writes but continues on failure
- Observer and Actor channels function normally

## Installation

**MongoDB is required only if you want persistent Story storage or debugging streams.**

### Linux/WSL Installation

```
# Ubuntu/Debian
sudo apt-get install -y mongodb-org

# Start MongoDB service
sudo systemctl start mongod
sudo systemctl enable mongod  # Auto-start on boot

# Verify MongoDB is running
sudo systemctl status mongod
```

**Note:** If you install MongoDB into WSL, after reboots it won't be running on Windows installations until at least one WSL prompt has been opened!

To enable MongoDB to be available from Windows, `bindIp:` below must be changed from 127.0.0.1 to 0.0.0.0 in `/etc/mongod.conf` then MongoDB must be restarted. The listing below shows the change to the network section of the configuration file.

```
# /etc/mongod.conf

# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/

...

# network interfaces
net:
  port: 27017
  bindIp: 0.0.0.0

...
```

### macOS Installation

```
# Using Homebrew
brew tap mongodb/brew
brew install mongodb-community

# Start MongoDB service
brew services start mongodb-community

# Verify MongoDB is running
brew services list | grep mongodb
```

### Windows Installation

1. Download MongoDB Community Server from mongodb.com
2. Run installer (choose "Complete" installation)
3. Install as Windows Service (recommended)
4. MongoDB runs automatically on startup

### Verify Installation (All Platforms):

```
# Connect to MongoDB shell
mongosh

# Or test connection
mongosh --eval "db.adminCommand('ping')"
```

Default MongoDB runs on `mongodb://localhost:27017`

## MongoDB Configuration

AIBE configuration is stored in `~/.AIBE/config.json`:

```
{
  "database": {
    "connection_string": "mongodb://localhost:27017/AIBE",
    "database_name": "AIBE",
```

```
    "collection_name": "Stories",
    "connection_timeout": 30,
    "max_pool_size": 100
  },
  "server": {
    "host": "localhost",
    "port": 3001,
    "debug": false,
    "log_level": "INFO"
  },
  "test_streaming": {
    "enabled": true,
    "database": "test_streams",
    "collection_prefix": "test_",
    "cleanup_after_days": 7
  },
  "server_streaming": {
    "enabled": true,
    "database": "server_streams",
    "collection_prefix": "server_",
    "cleanup_after_days": 7,
    "stream_observer": true,
    "stream_actor": true,
    "stream_story": true,
    "stream_log": true
  },
  "auth": {
    "config_password": "admin123"
  }
}
```

## Configuration Sections:

**database:**

- `connection_string`: MongoDB connection URL
- `database_name`: Database for Story storage (default: "AIBE")
- `collection_name`: Collection for Stories (default: "Stories")

- `connection_timeout`: Connection timeout in seconds
- `max_pool_size`: Maximum connection pool size

**server:**

- `host`: Server host (default: "localhost")
- `port`: Server port (default: 3001)
- `debug`: Enable debug mode
- `log_level`: Logging verbosity (ERROR, WARN, INFO, DEBUG)

**test_streaming:**

- `enabled`: Enable test event streaming to MongoDB
- `database`: Database for test streams
- `collection_prefix`: Prefix for test collection names
- `cleanup_after_days`: Auto-delete old test streams

**server_streaming:**

- `enabled`: Enable server event streaming for debugging
- `database`: Database for server event streams
- `collection_prefix`: Prefix for server collection names
- `stream_observer`: Log Observer channel events
- `stream_actor`: Log Actor channel commands
- `stream_story`: Log Story Assembly updates
- `stream_log`: Log server logs
- `cleanup_after_days`: Auto-delete old streams

**auth:**

- `config_password`: Password for config management endpoints

### Web-Based Configuration Editor

For convenience, AIBE provides a web interface to view and edit the configuration file at `http://localhost:3001/config` (credentials: `admin`/`admin123`). This is particularly useful for quickly adjusting MongoDB settings without manually editing the JSON file.

See **Server Management > Configuration > Web Configuration Interface** for complete details on using the web editor.

## Generating Sample Data

Before exploring MongoDB queries, you'll need browser interaction data stored in the database. The easiest way to generate comprehensive sample data is using the automated test suite.

### Prerequisites

Before running tests, ensure your environment is properly configured:

1. **AIBE Server Running:**

   ```
   aibe-server status
   # Should show: Server running on http://localhost:3001
   ```

   If not running: `aibe-server start`

2. **Chrome Extension Installed:**
   - Extension must be loaded in Chrome (Developer Mode)
   - Click extension icon - should show "Connected" status
   - If not connected, check server status and reload extension
3. **Chrome Browser Open:**
   - Open Chrome and navigate to `http://localhost:3001`. The AIBE home page should display.
   - Server root page confirms extension can communicate with server
   - Keep this browser window open and visible during tests
4. **Browser Tab Visible:**
   - **Critical:** Chrome throttles hidden/background tabs

- Tests will hang if browser tab is minimized or hidden
- Keep the test browser tab active and visible throughout test execution

## Using the Console Test Runner

The console test runner executes 48 automated tests across 12 test suites, creating realistic browser interaction Stories in MongoDB:

```
# From the server directory
cd server/aibe_server/tests

# Run all tests (creates Stories with diverse interactions)
node console-runner.js

# Optional: Use INFO log level for cleaner output
node console-runner.js --log-level=INFO
```

**What this creates:**

- Multiple Stories (one per test session)
- Paragraphs visiting localhost test pages
- Sentences showing page navigation patterns
- Words capturing form interactions, dropdown selections, button clicks
- Complete event histories with screen_status snapshots

**Test coverage includes:**

- Text input fields (username, email, password)
- Dropdowns (single and multi-select)
- Checkboxes and radio buttons
- Textareas and various input types
- Buttons and links
- Nested form controls

### Verify Data Was Created

After running tests, verify Stories were stored in MongoDB:

```
// In mongosh
use AIBE

// Count total stories
db.Stories.countDocuments({})

// View a sample story
db.Stories.findOne()

// Check what domains were visited
db.Stories.distinct("paragraphs.domain")
// Should show: ["localhost"]
```

### Alternative: Manual Browser Interaction

You can also generate data through normal browser usage:

1. Ensure AIBE server is running: `aibe-server status`
2. Open Chrome with the extension installed
3. Navigate to any websites and interact with forms
4. Each browser tab creates one Story
5. Check MongoDB: `db.Stories.countDocuments({})`

## Usage Examples

**Note:** The following examples demonstrate MongoDB queries using the Stories data structure. Use the test runner above to generate sample data, then explore these queries in `mongosh` or MongoDB Compass.

### Query All Stories from Today

```
// In mongosh
use AIBE

// Get all stories from today
const today = new Date();
today.setHours(0, 0, 0, 0);

db.Stories.find({
  "paragraphs.sentences.words.events.timestamp": {
    $gte: today.toISOString()
  }
})

// Count stories from today
db.Stories.countDocuments({
  "paragraphs.sentences.words.events.timestamp": {
    $gte: today.toISOString()
  }
})
```

### Find Stories by Domain

```
// Find all stories that visited github.com
db.Stories.find({
  "paragraphs.domain": "github.com"
})

// Find stories that visited multiple specific domains
db.Stories.find({
  "paragraphs.domain": {
    $in: ["github.com", "stackoverflow.com",
"developer.mozilla.org"]
  }
})

// Get unique list of all domains visited across all stories
db.Stories.distinct("paragraphs.domain")
```

```
// Count how many stories visited each domain
db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $group: {
    _id: "$paragraphs.domain",
    count: { $sum: 1 }
  }},
  { $sort: { count: -1 } }
])
```

### Extract Words Targeting Specific Controls

```
// Find all Words where user interacted with login/username fields
db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $unwind: "$paragraphs.sentences" },
  { $unwind: "$paragraphs.sentences.words" },
  { $match: {
    $or: [
      { "paragraphs.sentences.words.events.target.label":
/username/i },
      { "paragraphs.sentences.words.events.target.label": /email/i
},
      { "paragraphs.sentences.words.events.target.label": /login/i
}
    ]
  }},
  { $project: {
    session_id: 1,
    domain: "$paragraphs.domain",
    url: "$paragraphs.sentences.url",
    word: "$paragraphs.sentences.words"
  }}
])
// Find all password field interactions with final values
.db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $unwind: "$paragraphs.sentences" },
```

```
  { $unwind: "$paragraphs.sentences.words" },
  { $match: {
    "paragraphs.sentences.words.events.type": "keyboard",
    "paragraphs.sentences.words.events.target.label": /password/i
  }},
  { $project: {
    session_id: 1,
    url: "$paragraphs.sentences.url",
    target_label:
"$paragraphs.sentences.words.events.target.label",
    timestamp: "$paragraphs.sentences.words.events.timestamp"
  }}
])

// Extract all button clicks across all stories
.db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $unwind: "$paragraphs.sentences" },
  { $unwind: "$paragraphs.sentences.words" },
  { $unwind: "$paragraphs.sentences.words.events" },
  { $match: {
    "paragraphs.sentences.words.events.type": "mouse"
  }},
  { $project: {
    session_id: 1,
    domain: "$paragraphs.domain",
    button_label:
"$paragraphs.sentences.words.events.target.label",
    timestamp: "$paragraphs.sentences.words.events.timestamp"
  }},
  { $sort: { timestamp: -1 } }
])
])
```

### Analyze Patterns Across Sessions

```
// Find common URL navigation patterns
db.Stories.aggregate([
  { $unwind: "$paragraphs" },
```

```
  { $unwind: "$paragraphs.sentences" },
  { $group: {
    _id: {
      domain: "$paragraphs.domain",
      url: "$paragraphs.sentences.url"
    },
    visit_count: { $sum: 1 },
    sessions: { $addToSet: "$session_id" }
  }},
  { $project: {
    domain: "$_id.domain",
    url: "$_id.url",
    visit_count: 1,
    unique_sessions: { $size: "$sessions" }
  }},
  { $sort: { visit_count: -1 } },
  { $limit: 20 }
])

// Analyze form completion patterns - find which fields users fill
most often
db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $unwind: "$paragraphs.sentences" },
  { $unwind: "$paragraphs.sentences.words" },
  { $unwind: "$paragraphs.sentences.words.events" },
  { $match: {
    "paragraphs.sentences.words.events.type": "keyboard"
  }},
  { $group: {
    _id: {
      domain: "$paragraphs.domain",
      field_label:
"$paragraphs.sentences.words.events.target.label"
    },
    interaction_count: { $sum: 1 }
  }},
  { $sort: { interaction_count: -1 } },
  { $limit: 20 }
])
```

```
// Find navigation flow patterns (domain sequences)
db.Stories.aggregate([
  { $project: {
    session_id: 1,
    domain_sequence: "$paragraphs.domain"
  }},
  { $group: {
    _id: "$domain_sequence",
    count: { $sum: 1 },
    example_session: { $first: "$session_id" }
  }},
  { $sort: { count: -1 } },
  { $limit: 10 }
])

// Calculate average session length (number of paragraphs/domains
visited)
db.Stories.aggregate([
  { $project: {
    session_id: 1,
    paragraph_count: { $size: "$paragraphs" }
  }},
  { $group: {
    _id: null,
    avg_domains_per_session: { $avg: "$paragraph_count" },
    min_domains: { $min: "$paragraph_count" },
    max_domains: { $max: "$paragraph_count" }
  }}
])
```

## Managing Large Nested Data with Projections

Story documents contain deeply nested data (Story → Paragraphs → Sentences → Words → Events) and can become very large. Use MongoDB projections to control which fields are returned.

### Exclude Heavy Fields (Suppressing Data)

Exclude large fields you don't need to reduce document size and improve query performance:

```
// Exclude screen_status from all Words (can be very large with
full DOM snapshots)
db.Stories.find(
  {},
  {
    "paragraphs.sentences.words.screen_status": 0  // 0 = exclude
this field
  }
)

// Exclude multiple heavy fields
db.Stories.find(
  {},
  {
    "paragraphs.sentences.words.screen_status": 0,
    "paragraphs.sentences.words.events.target": 0,  // Exclude
detailed target info
    "_id": 0  // Exclude MongoDB's internal ID
  }
)

// Exclude all event data, keep only Word structure
db.Stories.aggregate([
  { $project: {
    session_id: 1,
    "paragraphs.domain": 1,
    "paragraphs.sentences.url": 1,
    "paragraphs.sentences.words.screen_status": 0,
    "paragraphs.sentences.words.events": 0
  }}
])
```

### Select Specific Fields (Including Only What You Need)

Return only the fields you actually need for analysis:

```
// Get only session IDs and domains visited
db.Stories.find(
  {},
  {
    session_id: 1,  // 1 = include this field
    "paragraphs.domain": 1,
    _id: 0  // Exclude MongoDB ID
  }
)

// Get only URLs visited and timestamps (no event details)
db.Stories.aggregate([
  { $project: {
    session_id: 1,
    urls: "$paragraphs.sentences.url",
    timestamps: "$paragraphs.sentences.words.events.timestamp"
  }}
])

// Extract only element labels that users interacted with
db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $unwind: "$paragraphs.sentences" },
  { $unwind: "$paragraphs.sentences.words" },
  { $unwind: "$paragraphs.sentences.words.events" },
  { $project: {
    session_id: 1,
    domain: "$paragraphs.domain",
    url: "$paragraphs.sentences.url",
    element_label:
"$paragraphs.sentences.words.events.target.label",
    event_type: "$paragraphs.sentences.words.events.type",
    timestamp: "$paragraphs.sentences.words.events.timestamp",
    _id: 0
  }},
  { $limit: 100 }
```

```
])
```

### Handle Varying Field Paths

Story structure can vary - some Words have events, some don't; some have screen_status, some don't:

```
// Use aggregation with $ifNull to handle missing fields
db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $unwind: "$paragraphs.sentences" },
  { $unwind: "$paragraphs.sentences.words" },
  { $project: {
    session_id: 1,
    domain: "$paragraphs.domain",
    url: "$paragraphs.sentences.url",
    // Safely access potentially missing fields
    has_events: {
      $cond: {
        if: { $gt: [{ $size: { $ifNull:
["$paragraphs.sentences.words.events", []] }}, 0] },
        then: true,
        else: false
      }
    },
    has_screen_status: {
      $cond: {
        if: { $ifNull:
["$paragraphs.sentences.words.screen_status", false] },
        then: true,
        else: false
      }
    },
    event_count: { $size: { $ifNull:
["$paragraphs.sentences.words.events", []] }}
  }}
])

// Filter to only Words that have events
```

```
db.Stories.aggregate([
  { $unwind: "$paragraphs" },
  { $unwind: "$paragraphs.sentences" },
  { $unwind: "$paragraphs.sentences.words" },
  { $match: {
    "paragraphs.sentences.words.events": { $exists: true, $ne: [] }
  }},
  { $project: {
    session_id: 1,
    domain: "$paragraphs.domain",
    events: "$paragraphs.sentences.words.events"
  }}
])
```

### Practical Example: Lightweight Session Summary

Get session overview without downloading massive event details:

```
// Compact session summary (domains, URLs, element counts - no
event details)
db.Stories.aggregate([
  { $project: {
    session_id: 1,
    total_paragraphs: { $size: "$paragraphs" },
    domains: "$paragraphs.domain",
    sentences_per_paragraph: {
      $map: {
        input: "$paragraphs",
        as: "para",
        in: { $size: "$$para.sentences" }
      }
    },
    urls_visited: {
      $reduce: {
        input: "$paragraphs.sentences.url",
        initialValue: [],
        in: { $concatArrays: ["$$value", ["$$this"]] }
      }
    },
```

```
    _id: 0
  }}
])

// Result is ~100x smaller than full document with all events and
screen_status
```

**Key Principles:**

- **Exclude first** if you know most fields aren't needed: `{ field: 0 }`
- **Include specific** if you need only a few fields: `{ field: 1 }`
- **Can't mix** include (1) and exclude (0) except for `_id`
- Use **aggregation pipelines** for complex field selection with transformations
- Always use **projections** when working with production data to reduce network/memory overhead

## JavaScript API Examples (MongoDB Node.js Driver)

The examples above use `mongosh` (interactive shell). For programmatic access (like AI systems), use the MongoDB Node.js driver.

**Note:** The following JavaScript examples follow standard MongoDB Node.js driver patterns and syntax. While they demonstrate correct API usage, they should be tested with your specific data structure before production use. The query logic matches the mongosh examples above.

```
// Install: npm install mongodb
const { MongoClient } = require('mongodb');

async function queryStories() {
  const client = new MongoClient('mongodb://localhost:27017');

  try {
    await client.connect();
    const db = client.db('AIBE');
    const stories = db.collection('Stories');
```

```
    // Example 1: Find stories with projection (exclude heavy
fields)
    const lightweightStories = await stories.find(
      {},
      {
        projection: {
          'paragraphs.sentences.words.screen_status': 0,  //
Exclude
          'paragraphs.sentences.words.events.target': 0
        }
      }
    ).toArray();

    // Example 2: Aggregation pipeline (select specific fields)
    const sessionSummaries = await stories.aggregate([
      { $project: {
        session_id: 1,
        domains: '$paragraphs.domain',
        total_paragraphs: { $size: '$paragraphs' },
        _id: 0
      }}
    ]).toArray();

    // Example 3: Filter and project in one query
    const todayStories = await stories.find(
      {
        'paragraphs.sentences.words.events.timestamp': {
          $gte: new Date().toISOString()
        }
      },
      {
        projection: {
          session_id: 1,
          'paragraphs.domain': 1,
          _id: 0
        }
      }
    ).toArray();
```

```
    // Example 4: Extract element interactions (complex
aggregation)
    const interactions = await stories.aggregate([
      { $unwind: '$paragraphs' },
      { $unwind: '$paragraphs.sentences' },
      { $unwind: '$paragraphs.sentences.words' },
      { $unwind: '$paragraphs.sentences.words.events' },
      { $match: {
        'paragraphs.sentences.words.events.type': 'keyboard'
      }},
      { $project: {
        session_id: 1,
        domain: '$paragraphs.domain',
        url: '$paragraphs.sentences.url',
        element_label:
'$paragraphs.sentences.words.events.target.label',
        timestamp: '$paragraphs.sentences.words.events.timestamp',
        _id: 0
      }},
      { $limit: 100 }
    ]).toArray();

    console.log(`Found ${interactions.length} keyboard
interactions`);

    return interactions;

  } finally {
    await client.close();
  }
}

// Example 5: Streaming large result sets (memory efficient)
async function streamStories() {
  const client = new MongoClient('mongodb://localhost:27017');

  try {
    await client.connect();
    const db = client.db('AIBE');
    const stories = db.collection('Stories');
```

```
    // Use cursor for large datasets instead of toArray()
    const cursor = stories.find(
      {},
      {
        projection: {
          session_id: 1,
          'paragraphs.domain': 1,
          _id: 0
        }
      }
    );

    // Process one document at a time
    for await (const story of cursor) {
      // Process each story without loading all into memory
      console.log(`Session: ${story.session_id}, Domains:
${story.paragraphs?.map(p => p.domain)}`);
    }

  } finally {
    await client.close();
  }
}

// Example 6: Conditional field handling (handle missing fields)
async function safeFieldAccess() {
  const client = new MongoClient('mongodb://localhost:27017');

  try {
    await client.connect();
    const db = client.db('AIBE');
    const stories = db.collection('Stories');

    const results = await stories.aggregate([
      { $unwind: '$paragraphs' },
      { $unwind: '$paragraphs.sentences' },
      { $unwind: '$paragraphs.sentences.words' },
      { $project: {
        session_id: 1,
```

```
        domain: '$paragraphs.domain',
        // Safely access potentially missing fields
        has_events: {
          $cond: {
            if: { $gt: [{ $size: { $ifNull:
['$paragraphs.sentences.words.events', []] }}, 0] },
            then: true,
            else: false
          }
        },
        event_count: { $size: { $ifNull:
['$paragraphs.sentences.words.events', []] }},
        _id: 0
      }}
    ]).toArray();

    return results;

  } finally {
    await client.close();
  }
}

// Run examples
queryStories().catch(console.error);
```

**Key Differences:**

| FEATURE | MONGOSH (SHELL) | JAVASCRIPT (NODE.JS DRIVER) |
|---|---|---|
| **Use case** | Interactive exploration | Programmatic AI access |
| **Syntax** | Direct commands: `db.Stories.find({})` | Async API: `await stories.find({}).toArray()` |
| **Results** | Auto-displayed | Must call `.toArray()` or iterate cursor |
| **Memory** | Good for small queries | Use cursors for large datasets |
| **AI Integration** | Manual copy/paste | Direct programmatic control |

**For AI Systems:** Use the JavaScript driver with cursor-based streaming for memory-efficient processing of large Story collections.

## Troubleshooting

### Connection Refused Errors

**Symptoms:**

```
ERROR: Failed to connect to MongoDB: [Errno 111] Connection refused
ERROR: MongoDB unavailable - Story Assembly disabled
```

**Solutions:**

1. **Verify MongoDB is running:**

```
# Linux/WSL
sudo systemctl status mongod

# macOS
brew services list | grep mongodb

# Windows - Check Services app
# Look for "MongoDB Server" service
```

2. **Start MongoDB if not running:**

```
# Linux/WSL
sudo systemctl start mongod

# macOS
brew services start mongodb-community

# Windows - Use Services app or:
net start MongoDB
```

3. **Check MongoDB is listening on default port:**

```
# Should show mongod listening on 27017
netstat -an | grep 27017
# Or
lsof -i :27017
```

4. **Test connection manually:**

```
mongosh --eval "db.adminCommand('ping')"
# Should return: { ok: 1 }
```

5. **Check firewall settings:**
   - Ensure localhost connections are allowed
   - Port 27017 should not be blocked for local connections

## Authentication Issues

**Symptoms:**

```
ERROR: Authentication failed for MongoDB
ERROR: MongoServerError: auth failed
```

**Solutions:**

1. **Check if authentication is enabled:**
   - Default MongoDB installation has authentication disabled
   - AIBE uses default connection without auth: `mongodb://localhost:27017/AIBE`
2. **If you enabled authentication on MongoDB:**

   Update `~/.AIBE/config.json` with credentials:

```
{
  "database": {
    "connection_string":
"mongodb://username:password@localhost:27017/AIBE",
    "database_name": "AIBE"
  }
}
```

3. **Create MongoDB user for AIBE (if needed):**

```
// In mongosh
use admin
db.createUser({
  user: "aibe_user",
  pwd: "your_secure_password",
  roles: [
    { role: "readWrite", db: "AIBE" },
    { role: "readWrite", db: "test_streams" },
    { role: "readWrite", db: "server_streams" }
  ]
})
```

4. **Test connection with credentials:**

```
mongosh
"mongodb://aibe_user:your_secure_password@localhost:27017/AIBE"
```

## Collection Not Found

**Symptoms:**

```
ERROR: Collection 'Stories' not found in database 'AIBE'
```

**Solutions:**

1. **Collections are created automatically:**
   - MongoDB creates collections automatically when first document is inserted
   - AIBE will create the Stories collection when you first use the browser extension
   - No manual collection creation needed
2. **Verify database exists:**

```
// In mongosh
show dbs
// Should list 'AIBE' if any data has been stored
```

3. **Check if collections exist:**

```
use AIBE
show collections
// Should show: Stories, and possibly
test_streams/server_streams
```

4. **If no data has been collected yet:**
    - Open a browser tab with the extension active
    - Navigate to any website and interact with it
    - Check server logs for "STORED NEW story to MongoDB" messages
    - Verify collection was created: `db.Stories.countDocuments({})`
5. **Manual collection creation (rarely needed):**

```
// In mongosh
use AIBE
db.createCollection("Stories")
```

## WSL-Specific Issues

**Symptoms:**

- MongoDB running in WSL but not accessible from Windows tools (MongoDB Compass, Robo 3T)
- Connection refused when connecting from Windows to WSL MongoDB
- `mongosh` works in WSL terminal but Windows applications can't connect

**Solutions:**

1. **Configure MongoDB to listen on all interfaces:**

   Edit `/etc/mongod.conf` in WSL:

```
# network interfaces
net:
  port: 27017
  bindIp: 0.0.0.0  # Change from 127.0.0.1 to listen on all
interfaces
```

2. **Restart MongoDB:**

```
sudo systemctl restart mongod
# Verify it restarted successfully
sudo systemctl status mongod
```

3. **Find WSL IP address:**

```
# In WSL terminal
hostname -I | awk '{print $1}'
# Example output: 172.24.123.45
```

4. **Connect from Windows MongoDB Compass:**
   - Use connection string: `mongodb://172.24.123.45:27017`
   - Replace IP with your WSL IP from step 3
   - This IP changes if WSL restarts, so check if connection fails
5. **Update AIBE config if needed:**

   If AIBE server is in WSL, it can continue using `localhost`:

```
{
  "database": {
    "connection_string": "mongodb://localhost:27017/AIBE"
  }
}
```

   If AIBE server runs on Windows but MongoDB in WSL:

```
{
  "database": {
    "connection_string":
"mongodb://172.24.123.45:27017/AIBE"
  }
}
```

**Security Note:** `bindIp: 0.0.0.0` allows connections from any network interface. For development on local machine this is safe. For production, use specific IP addresses or enable authentication.

### Performance Issues

**Symptoms:**

- Slow query response times
- High memory usage
- Server becomes unresponsive with large datasets

**Solutions:**

1. **Create indexes for common queries:**

```
// In mongosh
use AIBE

// Index on session_id for faster story lookups
db.Stories.createIndex({ "session_id": 1 })

// Index on domain for domain-based queries
db.Stories.createIndex({ "paragraphs.domain": 1 })

// Index on timestamps for time-based queries
db.Stories.createIndex({
"paragraphs.sentences.words.events.timestamp": 1 })

// Verify indexes
db.Stories.getIndexes()
```

2. **Limit result sizes for large queries:**

```
// Use .limit() to restrict results
db.Stories.find({}).limit(100)

// Use pagination for large datasets
db.Stories.find({}).skip(0).limit(50)
db.Stories.find({}).skip(50).limit(50)
```

3. **Use projection to return only needed fields:**

```
// Only return session_id and domain, not full documents
db.Stories.find(
  {},
  { session_id: 1, "paragraphs.domain": 1 }
)
```

4. **Monitor MongoDB performance:**

```
# Check current operations
mongosh --eval "db.currentOp()"

# Check server status
mongosh --eval "db.serverStatus()"
```

5. **Configure MongoDB memory settings:**
   - Edit `/etc/mongod.conf` (Linux) or MongoDB config file
   - Adjust `wiredTiger.engineConfig.cacheSizeGB` if needed
   - Default is 50% of RAM minus 1GB, usually sufficient

6. **Regular maintenance:**

```
// Clean up old test streams (if enabled)
use test_streams
db.dropDatabase()

use server_streams
db.dropDatabase()

// Compact collections to reclaim space
use AIBE
db.runCommand({ compact: "Stories" })
```

---

**Related Sections:**

- Story Assembly structure: See "Interpretation: Imposing Understanding on Browser Events"
- MongoDB storage format: See "Design Rationale: MongoDB Storage"

# Appendix -- Observer and Actor Event Schema

This appendix summarizes the canonical JSON schema for events flowing on the Observer and Actor channels. The key design principle is:

> ***Events describe what a human can see and do.***
>
> - `screen_status` *describes what is visible on the page.*
> - `mouse` *and* `keyboard` *events describe which visible element was acted upon and what its final state became.*

## Screen Status (Observer only)

`screen_status` events describe the stable contents of the current page as seen by a human user:

```
{
  "type": "screen_status",
  "url": "http://localhost:3001/test-controls",
  "focus_label": null,
  "focus_id": null,
  "visible_elements": [
    {
      "label": "Username",
      "control_type": "INPUT_TEXT",
      "type": "text",
      "value": "alice",
      "is_disabled": false,
      "has_focus": false
    },
```

```
    {
      "label": "Country",
      "control_type": "INPUT_DROPDOWN",
      "type": "select-one",
      "current_selection": "Select a country",
      "current_value": "",
      "clickable_options": [
        "\"United States\" → us",
        "\"Canada\" → ca"
      ],
      "is_open": false,
      "is_disabled": false,
      "href": null
    },
    {
      "label": "Internal Link",
      "control_type": "LINK",
      "href": "http://localhost:3001/test-result?action=internal",
      "is_disabled": false
    }
  ],
  "timestamp": "2025-07-03T18:23:25.698Z"
}
```


Notes:

- `visible_elements` contains human-facing semantics only: labels, control types, values, and (for links) `href`.
- Positional and DOM-internal fields are removed before data leaves the browser.

## Mouse Events

Mouse events describe clicks on visible elements.

### Observer → Server


```
{
  "type": "mouse",
  "event": "click",
  "target": {
    "label": "Internal Link",
    "value": {
      "Internal Link": "http://localhost:3001/test-result?
action=internal"
    }
  },
  "button": 0,
  "buttons": 0,
  "timestamp": "2025-07-03T18:23:30.037Z"
}
```


For buttons and links:

- `target.label` is the human-visible label.
- For links, `target.value` is a dictionary mapping the label to the `href`.
- The same `href` and `control_type: "LINK"` also appear in the corresponding `screen_status.visible_elements` entry.

For state-changing elements (checkboxes, radios, dropdowns), mouse events **also include** a `value` describing the final state, using the same dictionary conventions as keyboard events (see below). For elements without an obvious value (e.g., a plain button), `target.value` is an empty object `{}`.

### Actor → Browser

Actor uses the same shape for mouse commands:

```
{
  "type": "mouse",
  "event": "click",
  "target": {
    "label": "Internal Link",
    "value": {
      "Internal Link": "http://localhost:3001/test-result?
action=internal"
    }
  }
}
```

- Actor looks up the element by label in the latest `screen_status.visible_elements`.
- For links, it finds the `LINK` element with the matching `label` and calls the native `click()` method, letting the browser follow the `href`.

## Keyboard Events (Final-Value Semantics)

Keyboard events report **final control values**, not individual keystrokes.

### Text-like Controls

```
{
  "type": "keyboard",
  "event": "text_input",
  "target": {
    "label": "Email Address",
    "value": { "text": "user@example.com" }
  },
  "timestamp": "2025-01-08T10:15:35.500Z"
}
```

- Used for:
    - `INPUT_TEXT` (text, email, url, search, tel)
    - `INPUT_TEXTAREA`

    - `INPUT_PASSWORD` (when password values are visible)
- `value.text` is exactly what the user sees in the field at the end of the interaction.

## Dropdowns (Single-Select)

```
{
  "type": "keyboard",
  "event": "text_input",
  "target": {
    "label": "Country",
    "value": {
      "United States": "us"
    }
  }
}
```

- `value` is a dictionary mapping **display labels** to internal option values.
- Single-select dropdowns typically have one entry in the dictionary.

## Dropdowns (Multi-Select)

```
{
  "type": "keyboard",
  "event": "text_input",
  "target": {
    "label": "Skills (Multi-select)",
    "value": {
      "JavaScript": "js",
      "Python": "py"
    }
  }
}
```

- Multi-select dropdowns can have many entries; the keys are what the human sees.

### Checkboxes and Radio Buttons

```
{
  "type": "keyboard",
  "event": "text_input",
  "target": {
    "label": "Subscribe to newsletter",
    "value": {
      "Subscribe to newsletter": true
    }
  }
}
```

- For booleans, the dictionary maps the control label to `true` / `false`.
- The Actor uses this to set `.checked` on the underlying `<input>`.

## Symmetry: Observer and Actor

The Observer and Actor channels share the **same event schema**:

- Any Observer event of type `load`, `mouse`, or `keyboard` is a valid Actor command.
- Actor consumes events using the same structures that Observer produces:
    - `target.label` identifies the element on the page (via `screen_status.visible_elements`).
    - `target.value` uses the same dictionary/text conventions for final values.
- `screen_status` events are accepted by Actor but ignored (they are observational, not commands).

This symmetry means you can:

1. Record a browsing session via the Observer channel.
2. Save the events.
3. Replay them verbatim through the Actor channel to reproduce the behavior.