The CodeVideo Framework: Event Sourcing for IDE State Representation in Educational Software

Full Stack Craft LLC

January 18, 2025

Contents

1 Introduction **3**

2 CodeVideo Framework Architecture and Abstraction Layers **3**
- 2.1 Course . . . 3
- 2.2 Lesson . . . 3
- 2.3 Action . . . 4

3 Actions in Detail **4**
- 3.1 Action Composition . . . 5
- 3.2 Action Granularity . . . 5
- 3.3 Abstracted Actions . . . 6

4 Snapshots: Capturing the Full IDE State **6**
- 4.1 Reconstitution of IDE State . . . 7
- 4.2 Parallel Video Generation . . . 7

5 Validators: Deterministic Quality Assurance **7**

6 Renderers: Virtual Editors and CodeVideo Studio **7**

7 Recorders: Capturing Actions for Consumption in the CodeVideo Framework **8**
- 7.1 Mouse Recorder . . . 8
- 7.2 Keyboard Recorder . . . 8
- 7.3 Video Recorder . . . 8

8 Drivers: Playback Mechanisms for Automated Video Gen-

eration **8**
8.1 Temporal Abstract Syntax Tree Management 9
8.1.1 Formal Definition 9
8.1.2 Implementation Strategy 10

9 Overview of Framework Progress **10**

10 Applications For Extremely High Quality Dataset Generation For Use in Large Language Models **11**

11 Conclusion **11**

A Appendix A - Theoretical Performance of Parallel Video Generation **11**
A.1 Hardware Architecture and Capabilities 11
A.2 System Architecture 12
A.3 Performance Metrics 12
A.4 Scaling Characteristics 13

B Appendix B - Example JSON of a Simple Console Logging Lesson for Javascript **13**

C Appendix B - List of All Possible Actions within the CodeVideo Framework **15**

1 Introduction

The rapid growth of online education has transformed the software learning experience, creating unprecedented demand for high-quality, interactive software courses. As instructional video content rises, so do the demands for editing, structuring, and delivering this content at scale—a process that is both resource-intensive and technically challenging.

In this document, we present *CodeVideo*, an event-sourcing-based framework designed to streamline the creation and management of software instructional videos. By simulating every interaction (keystrokes and or mouse interactions) in an Integrated Development Environment (IDE), the framework can produce deterministic, repeatable sequences that simulate a live coding session. The result is not only the ability to generate a static snapshot of the IDE state at any given time but also the capability to generate entire instructional videos programmatically, ensuring consistency and precision across sessions.

2 CodeVideo Framework Architecture and Abstraction Layers

The CodeVideo Framework represents a modular and highly abstracted approach to IDE-based video course creation. This section provides an in-depth look at its primary abstractions and the structured hierarchy that enables reproducible, fine-grained state control over IDE interactions. The three main components are Courses, Lessons, and Actions.

2.1 Course

At the topmost level of the *CodeVideo* framework, we define a course, C, represented by:

- A unique identifier, C_{id}
- A descriptive name, C_{name}
- A sequence of instructional units or lessons, $L = \{L_1, L_2, \ldots, L_n\}$

2.2 Lesson

Each lesson, L, is composed of:

- An initial state snapshot of the project workspace, $S_{initial}$
- A sequence of discrete actions $A = \{a_1, a_2, \ldots, a_m\}$

- A final state snapshot S_{final} that serves as the baseline for subsequent lessons

2.3 Action

Central to the framework is the concept of an *action*, the smallest unit of interaction within the IDE. Actions, represented as a_i, encode the following elements:

$$a_i : \text{action\_type} \rightarrow \text{action\_value}$$

Examples of action types include text entry commands, cursor movements, file modifications, and verbal narration cues. For instance:

- **Text Entry Action:**

  a_{type} = "enter-text"
  a_{value} = "console.log('Hello, world!');"

- **Narration Action:**

  a_{type} = "speak-before"
  a_{value} = "I'm going to log a message to the console."

3 Actions in Detail

Actions form the foundation of the entire foundation of the CodeVideo framework. From an abstract standpoint, an action should be able to handle any basic interaction that is possible with a computer. There are limited ways of interacting with a computer, in order of most common to least common:

- Keyboard: A keyboard has a large variety of keys, but they can be broken down into 3 categories: letters, numbers, and symbols.
- Mouse: a mouse has motion sensor, 3 buttons, and a scroll wheel.
- Trackpad: which have 1 button and a touch surface.
- Audio inputs (microphones), which represent a series of points at varying amplitudes.
- Video inputs (cameras), which represent a series of images.

These methods comprise all the possible inputs to a computer.

We abstract these inputs to three main categories of actions: keyboard, mouse, and multimedia (audio and video).

Within an IDE, there are then 3 target locations where these actions can be applied: the editor, the file tree, and the terminal.

To distinguish the location of each action, the action name is suffixed with the target location. For example, a keystroke in the editor is represented as $type - editor$, a left click in the file tree is $left - click - file - tree$, and speaking about something in the terminal is $speak - terminal$.

Appendix A provides a sample of a series of actions, representing a Console Logging Lesson for JavaScript.

Appendix B provides all possible action names within the CodeVideo framework.

3.1 Action Composition

Actions are strictly composed of two parts: the action type and the action value. The action type is a string that describes the action, and the action value is a string that represents the value of the action. For example, an editor typing action would have the action type of "type-editor" and the action value of "console.log('Hello, world!');".

3.2 Action Granularity

The framework implements a hierarchical action granularity system, where the base unit of interaction can be configured according to course requirements:

- **Atomic Actions** (Default): The finest level of granularity, capturing:
  - Individual keystrokes: $a_{key} = \{key : `a', step : n\}$
  - Single cursor movements: $a_{cursor} = \{position : (x, y), step : n\}$
  - Mouse events: $a_{mouse} = \{type :' click', position : (x, y), step : n\}$
- **Composed Actions**: Higher-level abstractions that can be configured:
  - Line-level changes: $a_{line} = \{content : string, lineNumber : n\}$

- Block-level modifications: $a_{block} = \{content : string, range : (start, end)\}$
- Semantic operations: $a_{semantic} = \{type : `function-declaration', content : string\}$

The granularity level can be adjusted dynamically, with the framework maintaining the atomic actions internally while presenting a more abstract interface when desired:

$$A_{composed} = f(a_1, a_2, ..., a_n) \text{ where } a_i \text{ are atomic actions}$$

3.3 Abstracted Actions

The notion of an "Abstracted Action" is a highly composed action that represents, and can be broken down, into atomic actions. While the action name becomes simpler, the action value becomes more complex. For example, the "mouse" action is an abstracted action for an entire mouse movement, including the mouse's x and y coordinates, timestamp, what buttons were clicked, and so on.

4 Snapshots: Capturing the Full IDE State

Snapshots represent the entire state of the IDE at a given point, allowing for precise rollback and reproducibility:

$$\begin{aligned} S = \langle & \text{metadata}, \\ & \text{file\_structure}, \\ & \text{selected\_file}, \\ & \text{open\_files}, \\ & \text{editor\_content}, \\ & \text{caret\_position}, \\ & \text{terminal\_content} \rangle \end{aligned}$$

Where:

- **Metadata** contains high-level course information.
- **File Structure** denotes the directory tree visible in the IDE.
- **Selected File** and **Open Files** track active files.
- **Editor Content** and **Caret Position** record the current text and cursor position.
- **Terminal Content** displays command-line output.

4.1 Reconstitution of IDE State

In a software lesson, the authors, keyboard, and mouse components can all be used in parallel while creating lesson, much like how a real software developer would interact with an IDE. The IDE then, is the more or less linear result of these interactions. The IDE state can be reconstituted by replaying the actions in the same order they were recorded. We use logical time to represent the start of any one of these "parallel" actions, and let the renderers handle the overlapping complexity.

4.2 Parallel Video Generation

With a 100% reconstituable IDE state, the framework can generate multiple videos in parallel and stitch them together flawless at the end. For example, consider a 200 step lesson. After validating that the state is continuous between each step, the framework can generate 200 short videos in parallel, each video representing a step in the lesson. The videos can then be stitched together to create a single video that represents the entire lesson. Because the IDE state is guaranteed by the framework to be continuous between each step, the final video will be a seamless representation of the entire lesson. See Appendix A for a theoretical performance analysis of parallel video generation.

5 Validators: Deterministic Quality Assurance

Each lesson's continuity is verified through a *snapshot validator*, to confirm that the content of the given project at the end of one lesson matches the initial state of the next lesson:

$$\forall i \in [1, n-1], \quad S_{final}(L_i) = S_{initial}(L_{i+1})$$

If any two consecutive snapshots do not match, the validator throws an error, ensuring seamless progression across lessons.

6 Renderers: Virtual Editors and CodeVideo Studio

The CodeVideo framework integrates a *virtual editor*, which emulates IDE interactions in a standalone environment:

- The virtual editor provides snapshots on demand via $getProjectSnapshot()$, allowing course authors to generate real-time previews of each state.

The rendering engine, *CodeVideo Studio* (https://studio.codevideo.io), provides a visual representation within the IDE, enabling authors to edit actions dynamically and validate course accuracy.

7 Recorders: Capturing Actions for Consumption in the CodeVideo Framework

The framework supports a range of recorders.

7.1 Mouse Recorder

CodeVideo offers 'codevideo-mouse' library as a rich TypeScript React component that captures mouse interactions within the IDE.

7.2 Keyboard Recorder

The first record directly records mouse, keystrokes, and microphone audio:

- Keystrokes and cursor movements are recorded in the main editor.
- Interactions within the file tree, terminal, and narration windows are recorded and transformed to action format to build the repeatable and full IDE state representation.

7.3 Video Recorder

The second is visual-based, capturing the entire screen and using computer vision to detect and interpret user interactions:

- The visual recorder captures the entire screen, including the IDE and any additional windows.
- The video is broken into individual frames, and computer vision algorithms detect and interpret user interactions.
- The detected interactions are transformed into actions and used to build the repeatable and full IDE state representation.

8 Drivers: Playback Mechanisms for Automated Video Generation

CodeVideo provides multiple playback mechanisms for automated video generation:

- **CodeVideo Studio** our main editor, based on a Monaco editor along side an action editor. Includes time travel and snapshot validation.
- **Monaco Editor Driver** for localhost single editor-based video generation (beta)
- **VSCode Driver** for localhost complete IDE editor-based video generation (beta)

8.1 Temporal Abstract Syntax Tree Management

Central to the framework's refactoring capabilities is the concept of a Temporal Abstract Syntax Tree (TAST), which extends traditional AST manipulation to include logical time as a dimension. Simply described, this is the abstract syntax tree after every moment (action) in time. The concept of TASTs are not new, and have been explored in various works [**moore2022temporal**].

8.1.1 Formal Definition

A Temporal AST node at logical step n is defined as:

$$TAST_{node}(n) = \langle ID, Type, Value(n), Children(n), Metadata(n) \rangle$$

Where:

- ID is a unique identifier persisting across the action sequence
- $Type$ represents the node type (e.g., Identifier, FunctionDeclaration)
- $Value(n)$ is the node's value at step n
- $Children(n)$ represents child nodes at step n
- $Metadata(n)$ stores additional information like scope and references

When performing retrospective modifications (e.g., variable renaming), the framework uses a forward propagation algorithm:

$$\forall m > n : TAST(m) = Apply(TAST(m-1), Modification(n))$$

The modification propagation follows these principles:

1. **Identity Preservation**: Node identities remain constant across the sequence, enabling tracking of elements despite name changes:

$$ID(TAST_{node}(n)) = ID(TAST_{node}(m)) \quad \forall n, m$$

2. **Reference Integrity**: All references to modified nodes are updated while maintaining semantic correctness:

$$Refs(TAST_{node}(m)) = UpdateRefs(Refs(TAST_{node}(n)), Modification(n))$$

3. **Scope Resolution**: Modifications respect lexical scoping rules across the logical sequence:

$$Scope(TAST_{node}(m)) = ResolveScope(Scope(TAST_{node}(n)), Environment(m))$$

8.1.2 Implementation Strategy

The framework implements temporal AST modifications through a combination of:

- Forward replay with modification injection
- Lazy evaluation of affected subtrees
- Cached intermediate states for performance optimization

This approach enables complex refactoring operations while maintaining the integrity of the recorded action sequence. For example, when renaming a variable at step n, the system:

1. Captures the modification intent: $M = \{type : "rename", target : ID, newValue : name\}$
2. Identifies all subsequent steps containing references to the target: $Steps_{affected} = \{m | m \geq n \wedge HasRef(TAST(m), ID)\}$
3. Applies the modification across all affected steps while preserving other concurrent modifications

9 Overview of Framework Progress

The environments and actions to make codevideos form an interesting matrix of thte software libraries and progress of the entire framework as a whole. The following table reflect the current state of the framework, when looking at the various environments and their Renderers, Recorders, and Drivers, as well as what language the package is written in.

Environment	Renderers	Recorders	Drivers
Web	CodeVideo Studio	Keyboard	CodeVideo Stu
Visual Studio Code	Visual Studio Code's GUI	'codevideo-vs-code-extension'	'codevideo-vs-c

Note within these, the underlying event source architecture of the snapshot generators via the virtual items is the same, and the drivers are the only thing that changes.

10 Applications For Extremely High Quality Dataset Generation For Use in Large Language Models

A large component of context missing from large language models and transformers is this exact travel-through-time context. By including the full correct and complete steps to creating software, either through the array of actions or a TAST to LLM training data, we can provide a more comprehensive dataset for coding training data.

11 Conclusion

The *CodeVideo* framework offers a robust, reproducible method for generating IDE-based educational content, reducing manual editing and increasing course consistency. With its event-sourcing architecture, the framework not only captures precise IDE states but also provides a platform for creating interactive, context-rich programming tutorials.

A Appendix A - Theoretical Performance of Parallel Video Generation

The CodeVideo framework leverages modern hardware architecture to achieve exceptional performance through massive parallelization. This section analyzes performance metrics based on the Apple Silicon M4 architecture with 128GB unified memory.

A.1 Hardware Architecture and Capabilities

The framework's performance is optimized for modern unified memory architectures, particularly benefiting from:

- **Unified Memory:** 128GB shared between CPU and GPU eliminates memory transfer overhead

- **Memory Bandwidth:** 546GB/second enables rapid frame buffer processing
- **Hardware Video Encoders:** Native hardware acceleration for concurrent video streams
- **Parallel Chrome Instances:** Support for 80-100 concurrent headless browsers

A.2 System Architecture

The framework employs a multi-tiered execution model:

- **Orchestration Layer:** Written in Rust/Go for optimal system resource management
- **Rendering Engine:** Headless Chrome instances managed via Puppeteer
- **Video Processing:** Hardware-accelerated encoding via VideoToolbox
- **Storage Layer:** NVMe SSD array for parallel I/O operations

A.3 Performance Metrics

Table 2 compares CodeVideo generation times with traditional video editing workflows across various lesson sizes. These estimates assume:

- Average step duration of 4 seconds
- 90 parallel Chrome instances (optimized for M4 architecture)
- Hardware-accelerated video encoding
- Traditional editing requiring 2.5x video length plus setup time per step

Number of Steps	**Total Video Length** *(minutes)*	**CodeVideo Generation** *(seconds)*	**Traditional Editing Time** *(hours)*	**Performance Improvement** *(factor)*
50	3.3	17	0.56	5,324x
100	6.7	34	1.11	5,324x
200	13.3	51	2.22	7,098x
500	33.3	102	5.56	8,873x

Table 2: Performance comparison between CodeVideo and traditional video editing workflows on M4 architecture

A.4 Scaling Characteristics

The framework exhibits near-linear scaling up to the parallel instance limit, with performance primarily bounded by:

- **I/O Operations:** NVMe bandwidth for intermediate file handling
- **Memory Management:** Dynamic allocation across Chrome instances
- **Video Encoding:** Hardware encoder queue depth

As demonstrated by the performance metrics, the CodeVideo framework achieves approximately a 200x improvement in production time compared to traditional video editing workflows when leveraging modern unified memory architecture. This extraordinary efficiency gain is particularly pronounced for larger lessons, making it feasible to generate entire courses of content in minutes rather than hours or days.

The parallel processing capabilities of the M4 architecture, combined with hardware-accelerated video encoding and the unified memory model, enable the framework to process approximately 90 steps simultaneously. This results in the remarkably fast generation times shown above, with even 500-step lessons completing in under 2 minutes.

B Appendix B - Example JSON of a Simple Console Logging Lesson for Javascript

```
[
    {
        "name": "speak-before",
        "value": "Today, we're going to learn about how to
        ↪  use the console.log function in JavaScript."
    },
    {
        "name": "speak-before",
        "value": "I've already got a hello-world.js file
        ↪  prepared here - let's open it up."
    },
    {
        "name": "click-filename",
        "value": "hello-world.js"
    },
    {
        "name": "click-editor",
```

```
    "value": "1"
},
{
    "name": "speak-before",
    "value": "Now, to log things to your console,
    ↪  simply make a call to the console.log function,
    ↪  passing in the text you want to log."
},
{
    "name": "type-editor",
    "value": "console.log('Hello, world!');"
},
{
    "name": "save",
    "value": "1"
},
{
    "name": "speak-before",
    "value": "Now we'll open up a terminal and run this
    ↪  file."
},
{
    "name": "open-terminal",
    "value": "1"
},
{
    "name": "click-terminal",
    "value": "1"
},
{
    "name": "type-terminal",
    "value": "node hello-world.js"
},
{
    "name": "enter",
    "value": "1"
},
{
    "name": "wait",
    "value": "2000"
},
{
    "name": "speak-before",
```

```
        "value": "And of course we get the expected output
        ↪  - 'hello world!' printed to the console."
    },
    {
        "name": "speak-before",
        "value": "And that's about it! You now know how to
        ↪  log things to your console in JavaScript!"
    }
]
```

C Appendix B - List of All Possible Actions within the CodeVideo Framework

Key Combination	Context	Code
editor Bindings		
‘	editor	`Backquote-editor`
Shift+{}	editor	`Shift+Backquote-editor`
1	editor	`Digit1-editor`
Shift+!	editor	`Shift+Digit1-editor`
2	editor	`Digit2-editor`
Shift+@	editor	`Shift+Digit2-editor`
3	editor	`Digit3-editor`
Shift+#	editor	`Shift+Digit3-editor`
4	editor	`Digit4-editor`
Shift+$	editor	`Shift+Digit4-editor`
5	editor	`Digit5-editor`
Shift+%	editor	`Shift+Digit5-editor`
6	editor	`Digit6-editor`
Shift+{}	editor	`Shift+Digit6-editor`
7	editor	`Digit7-editor`
Shift+&	editor	`Shift+Digit7-editor`
8	editor	`Digit8-editor`
Shift+*	editor	`Shift+Digit8-editor`
9	editor	`Digit9-editor`
Shift+(	editor	`Shift+Digit9-editor`
0	editor	`Digit0-editor`
Shift+)	editor	`Shift+Digit0-editor`
-	editor	`Minus-editor`
Shift+_	editor	`Shift+Minus-editor`
=	editor	`Equal-editor`
Shift++	editor	`Shift+Equal-editor`

Key Combination	Context	Code
q	editor	`KeyQ-editor`
Shift+Q	editor	`Shift+KeyQ-editor`
w	editor	`KeyW-editor`
Shift+W	editor	`Shift+KeyW-editor`
e	editor	`KeyE-editor`
Shift+E	editor	`Shift+KeyE-editor`
r	editor	`KeyR-editor`
Shift+R	editor	`Shift+KeyR-editor`
t	editor	`KeyT-editor`
Shift+T	editor	`Shift+KeyT-editor`
y	editor	`KeyY-editor`
Shift+Y	editor	`Shift+KeyY-editor`
u	editor	`KeyU-editor`
Shift+U	editor	`Shift+KeyU-editor`
i	editor	`KeyI-editor`
Shift+I	editor	`Shift+KeyI-editor`
o	editor	`KeyO-editor`
Shift+O	editor	`Shift+KeyO-editor`
p	editor	`KeyP-editor`
Shift+P	editor	`Shift+KeyP-editor`
[	editor	`BracketLeft-editor`
Shift+{	editor	`Shift+BracketLeft-editor`
]	editor	`BracketRight-editor`
Shift+}	editor	`Shift+BracketRight-editor`
\{}	editor	`Backslash-editor`
Shift+|	editor	`Shift+Backslash-editor`
a	editor	`KeyA-editor`
Shift+A	editor	`Shift+KeyA-editor`
s	editor	`KeyS-editor`
Shift+S	editor	`Shift+KeyS-editor`
d	editor	`KeyD-editor`
Shift+D	editor	`Shift+KeyD-editor`
f	editor	`KeyF-editor`
Shift+F	editor	`Shift+KeyF-editor`
g	editor	`KeyG-editor`
Shift+G	editor	`Shift+KeyG-editor`
h	editor	`KeyH-editor`
Shift+H	editor	`Shift+KeyH-editor`
j	editor	`KeyJ-editor`
Shift+J	editor	`Shift+KeyJ-editor`
k	editor	`KeyK-editor`

Key Combination	Context	Code
Shift+K	editor	`Shift+KeyK-editor`
l	editor	`KeyL-editor`
Shift+L	editor	`Shift+KeyL-editor`
;	editor	`Semicolon-editor`
Shift+:	editor	`Shift+Semicolon-editor`
’	editor	`Quote-editor`
Shift+”	editor	`Shift+Quote-editor`
z	editor	`KeyZ-editor`
Shift+Z	editor	`Shift+KeyZ-editor`
x	editor	`KeyX-editor`
Shift+X	editor	`Shift+KeyX-editor`
c	editor	`KeyC-editor`
Shift+C	editor	`Shift+KeyC-editor`
v	editor	`KeyV-editor`
Shift+V	editor	`Shift+KeyV-editor`
b	editor	`KeyB-editor`
Shift+B	editor	`Shift+KeyB-editor`
n	editor	`KeyN-editor`
Shift+N	editor	`Shift+KeyN-editor`
m	editor	`KeyM-editor`
Shift+M	editor	`Shift+KeyM-editor`
,	editor	`Comma-editor`
Shift+<	editor	`Shift+Comma-editor`
.	editor	`Period-editor`
Shift+>	editor	`Shift+Period-editor`
/	editor	`Slash-editor`
Shift+?	editor	`Shift+Slash-editor`
	editor	`Space-editor`
terminal Bindings		
‘	terminal	`Backquote-terminal`
Shift+{}	terminal	`Shift+Backquote-terminal`
1	terminal	`Digit1-terminal`
Shift+!	terminal	`Shift+Digit1-terminal`
2	terminal	`Digit2-terminal`
Shift+@	terminal	`Shift+Digit2-terminal`
3	terminal	`Digit3-terminal`
Shift+#	terminal	`Shift+Digit3-terminal`
4	terminal	`Digit4-terminal`
Shift+$	terminal	`Shift+Digit4-terminal`
5	terminal	`Digit5-terminal`

Key Combination	Context	Code
Shift+%	terminal	Shift+Digit5-terminal
6	terminal	Digit6-terminal
Shift+{}	terminal	Shift+Digit6-terminal
7	terminal	Digit7-terminal
Shift+&	terminal	Shift+Digit7-terminal
8	terminal	Digit8-terminal
Shift+*	terminal	Shift+Digit8-terminal
9	terminal	Digit9-terminal
Shift+(	terminal	Shift+Digit9-terminal
0	terminal	Digit0-terminal
Shift+)	terminal	Shift+Digit0-terminal
-	terminal	Minus-terminal
Shift+_	terminal	Shift+Minus-terminal
=	terminal	Equal-terminal
Shift++	terminal	Shift+Equal-terminal
q	terminal	KeyQ-terminal
Shift+Q	terminal	Shift+KeyQ-terminal
w	terminal	KeyW-terminal
Shift+W	terminal	Shift+KeyW-terminal
e	terminal	KeyE-terminal
Shift+E	terminal	Shift+KeyE-terminal
r	terminal	KeyR-terminal
Shift+R	terminal	Shift+KeyR-terminal
t	terminal	KeyT-terminal
Shift+T	terminal	Shift+KeyT-terminal
y	terminal	KeyY-terminal
Shift+Y	terminal	Shift+KeyY-terminal
u	terminal	KeyU-terminal
Shift+U	terminal	Shift+KeyU-terminal
i	terminal	KeyI-terminal
Shift+I	terminal	Shift+KeyI-terminal
o	terminal	KeyO-terminal
Shift+O	terminal	Shift+KeyO-terminal
p	terminal	KeyP-terminal
Shift+P	terminal	Shift+KeyP-terminal
[	terminal	BracketLeft-terminal
Shift+{	terminal	Shift+BracketLeft-terminal
]	terminal	BracketRight-terminal
Shift+}	terminal	Shift+BracketRight-terminal
\{}	terminal	Backslash-terminal
Shift+|	terminal	Shift+Backslash-terminal

Key Combination	Context	Code
a	terminal	`KeyA-terminal`
Shift+A	terminal	`Shift+KeyA-terminal`
s	terminal	`KeyS-terminal`
Shift+S	terminal	`Shift+KeyS-terminal`
d	terminal	`KeyD-terminal`
Shift+D	terminal	`Shift+KeyD-terminal`
f	terminal	`KeyF-terminal`
Shift+F	terminal	`Shift+KeyF-terminal`
g	terminal	`KeyG-terminal`
Shift+G	terminal	`Shift+KeyG-terminal`
h	terminal	`KeyH-terminal`
Shift+H	terminal	`Shift+KeyH-terminal`
j	terminal	`KeyJ-terminal`
Shift+J	terminal	`Shift+KeyJ-terminal`
k	terminal	`KeyK-terminal`
Shift+K	terminal	`Shift+KeyK-terminal`
l	terminal	`KeyL-terminal`
Shift+L	terminal	`Shift+KeyL-terminal`
;	terminal	`Semicolon-terminal`
Shift+:	terminal	`Shift+Semicolon-terminal`
’	terminal	`Quote-terminal`
Shift+”	terminal	`Shift+Quote-terminal`
z	terminal	`KeyZ-terminal`
Shift+Z	terminal	`Shift+KeyZ-terminal`
x	terminal	`KeyX-terminal`
Shift+X	terminal	`Shift+KeyX-terminal`
c	terminal	`KeyC-terminal`
Shift+C	terminal	`Shift+KeyC-terminal`
v	terminal	`KeyV-terminal`
Shift+V	terminal	`Shift+KeyV-terminal`
b	terminal	`KeyB-terminal`
Shift+B	terminal	`Shift+KeyB-terminal`
n	terminal	`KeyN-terminal`
Shift+N	terminal	`Shift+KeyN-terminal`
m	terminal	`KeyM-terminal`
Shift+M	terminal	`Shift+KeyM-terminal`
,	terminal	`Comma-terminal`
Shift+<	terminal	`Shift+Comma-terminal`
.	terminal	`Period-terminal`
Shift+>	terminal	`Shift+Period-terminal`
/	terminal	`Slash-terminal`

Key Combination	Context	Code
Shift+?	terminal	`Shift+Slash-terminal`
	terminal	`Space-terminal`
file-tree Bindings		
‘	file-tree	`Backquote-file-tree`
Shift+{}	file-tree	`Shift+Backquote-file-tree`
1	file-tree	`Digit1-file-tree`
Shift+!	file-tree	`Shift+Digit1-file-tree`
2	file-tree	`Digit2-file-tree`
Shift+@	file-tree	`Shift+Digit2-file-tree`
3	file-tree	`Digit3-file-tree`
Shift+#	file-tree	`Shift+Digit3-file-tree`
4	file-tree	`Digit4-file-tree`
Shift+$	file-tree	`Shift+Digit4-file-tree`
5	file-tree	`Digit5-file-tree`
Shift+%	file-tree	`Shift+Digit5-file-tree`
6	file-tree	`Digit6-file-tree`
Shift+{}	file-tree	`Shift+Digit6-file-tree`
7	file-tree	`Digit7-file-tree`
Shift+&	file-tree	`Shift+Digit7-file-tree`
8	file-tree	`Digit8-file-tree`
Shift+*	file-tree	`Shift+Digit8-file-tree`
9	file-tree	`Digit9-file-tree`
Shift+(	file-tree	`Shift+Digit9-file-tree`
0	file-tree	`Digit0-file-tree`
Shift+)	file-tree	`Shift+Digit0-file-tree`
-	file-tree	`Minus-file-tree`
Shift+_	file-tree	`Shift+Minus-file-tree`
=	file-tree	`Equal-file-tree`
Shift++	file-tree	`Shift+Equal-file-tree`
q	file-tree	`KeyQ-file-tree`
Shift+Q	file-tree	`Shift+KeyQ-file-tree`
w	file-tree	`KeyW-file-tree`
Shift+W	file-tree	`Shift+KeyW-file-tree`
e	file-tree	`KeyE-file-tree`
Shift+E	file-tree	`Shift+KeyE-file-tree`
r	file-tree	`KeyR-file-tree`
Shift+R	file-tree	`Shift+KeyR-file-tree`
t	file-tree	`KeyT-file-tree`
Shift+T	file-tree	`Shift+KeyT-file-tree`
y	file-tree	`KeyY-file-tree`

Key Combination	Context	Code
Shift+Y	file-tree	`Shift+KeyY-file-tree`
u	file-tree	`KeyU-file-tree`
Shift+U	file-tree	`Shift+KeyU-file-tree`
i	file-tree	`KeyI-file-tree`
Shift+I	file-tree	`Shift+KeyI-file-tree`
o	file-tree	`KeyO-file-tree`
Shift+O	file-tree	`Shift+KeyO-file-tree`
p	file-tree	`KeyP-file-tree`
Shift+P	file-tree	`Shift+KeyP-file-tree`
[	file-tree	`BracketLeft-file-tree`
Shift+{	file-tree	`Shift+BracketLeft-file-tree`
]	file-tree	`BracketRight-file-tree`
Shift+}	file-tree	`Shift+BracketRight-file-tree`
\{}	file-tree	`Backslash-file-tree`
Shift+|	file-tree	`Shift+Backslash-file-tree`
a	file-tree	`KeyA-file-tree`
Shift+A	file-tree	`Shift+KeyA-file-tree`
s	file-tree	`KeyS-file-tree`
Shift+S	file-tree	`Shift+KeyS-file-tree`
d	file-tree	`KeyD-file-tree`
Shift+D	file-tree	`Shift+KeyD-file-tree`
f	file-tree	`KeyF-file-tree`
Shift+F	file-tree	`Shift+KeyF-file-tree`
g	file-tree	`KeyG-file-tree`
Shift+G	file-tree	`Shift+KeyG-file-tree`
h	file-tree	`KeyH-file-tree`
Shift+H	file-tree	`Shift+KeyH-file-tree`
j	file-tree	`KeyJ-file-tree`
Shift+J	file-tree	`Shift+KeyJ-file-tree`
k	file-tree	`KeyK-file-tree`
Shift+K	file-tree	`Shift+KeyK-file-tree`
l	file-tree	`KeyL-file-tree`
Shift+L	file-tree	`Shift+KeyL-file-tree`
;	file-tree	`Semicolon-file-tree`
Shift+:	file-tree	`Shift+Semicolon-file-tree`
’	file-tree	`Quote-file-tree`
Shift+”	file-tree	`Shift+Quote-file-tree`
z	file-tree	`KeyZ-file-tree`
Shift+Z	file-tree	`Shift+KeyZ-file-tree`
x	file-tree	`KeyX-file-tree`
Shift+X	file-tree	`Shift+KeyX-file-tree`

Key Combination	Context	Code
c	file-tree	`KeyC-file-tree`
Shift+C	file-tree	`Shift+KeyC-file-tree`
v	file-tree	`KeyV-file-tree`
Shift+V	file-tree	`Shift+KeyV-file-tree`
b	file-tree	`KeyB-file-tree`
Shift+B	file-tree	`Shift+KeyB-file-tree`
n	file-tree	`KeyN-file-tree`
Shift+N	file-tree	`Shift+KeyN-file-tree`
m	file-tree	`KeyM-file-tree`
Shift+M	file-tree	`Shift+KeyM-file-tree`
,	file-tree	`Comma-file-tree`
Shift+<	file-tree	`Shift+Comma-file-tree`
.	file-tree	`Period-file-tree`
Shift+>	file-tree	`Shift+Period-file-tree`
/	file-tree	`Slash-file-tree`
Shift+?	file-tree	`Shift+Slash-file-tree`
	file-tree	`Space-file-tree`