The Complete Manual

Every menu, panel, control, and command

Runic 0.61.1 · build 73e0f01 · September 2026

A fast, native macOS editor for very large files.

The Complete Manual

Introduction

Runic is a native macOS text editor built for files that other editors refuse to open. This manual documents every menu, panel, control, and command in the application.

If you are new to Runic, read the *Quick User's Guide* first — it covers the same ground in a tenth of the length. This document is the reference you come back to.

How Runic differs, and why it matters to you

Three design decisions explain nearly every behaviour described in this manual.

Files are mapped, not loaded. Runic asks macOS to map the file into its address space and reads only the bytes it needs to draw the screen. Consequences you will notice: opening a 2 GB file takes about as long as opening a small one; memory use stays far below file size; and Runic never has to "finish loading" before you can work.

Work is proportional to what is visible. Drawing a frame costs the same whether the file is 2 KB or 2 GB. This is why scrolling stays smooth, and why features like syntax highlighting, folding, and bracket colouring do not slow down on large files — they only ever run on the visible region.

Long operations stream. Search, formatting, and saving pass over the file in chunks rather than assembling it in memory. You can pretty-print a JSON file larger than your RAM.

Where a feature behaves unusually, it is almost always one of these three at work, and this manual says so.

Conventions in this manual

- Menu paths are written **File › Save As…**
- Shortcuts appear as key chips: `⌘S`
- Command identifiers (used for remapping keys) appear as `file.save`
- Where a command is greyed out, the manual states the condition that enables it

---

Installation and updates

Installing

Download `Runic-<version>.dmg`, open it, and drag Runic to Applications. When Runic downloads an update itself, it verifies the package first and can install and relaunch after you approve it; if the install location cannot be replaced directly, it shows the downloaded DMG so you can install manually.

Runic is signed with a Developer ID Application certificate and notarized by Apple. The notarization ticket is *stapled* into the app, so the first launch succeeds even with no network connection and even on a Mac that has never run Runic.

System requirements

macOS 13 (Ventura) or later. The application is a universal binary containing native code for both Apple Silicon (arm64) and Intel (x86_64) Macs.

Checking your version

Runic › About Runic shows the version and the exact source commit the build was made from:

```
Version 0.57.0 (281cc34)
```

The value in parentheses identifies the build precisely. Two builds can share a version number; they cannot share a commit. If you are reporting a problem, include it.

Updates

Help › Check for Updates… (`help.checkForUpdates`) compares your installed version against the current release and tells you whether a newer one exists. Runic also checks quietly on launch; if a newer version exists, it shows a non-blocking banner instead of opening an alert.

The window

Layout

From top to bottom, a Runic window contains:

Header bar — the document's identity and state: file name, path, encoding, whether the document is remote, and a breadcrumb showing the structural position of your cursor (which object or function you are inside). For a remote document with no host agent installed, the header appends · **no agent**.

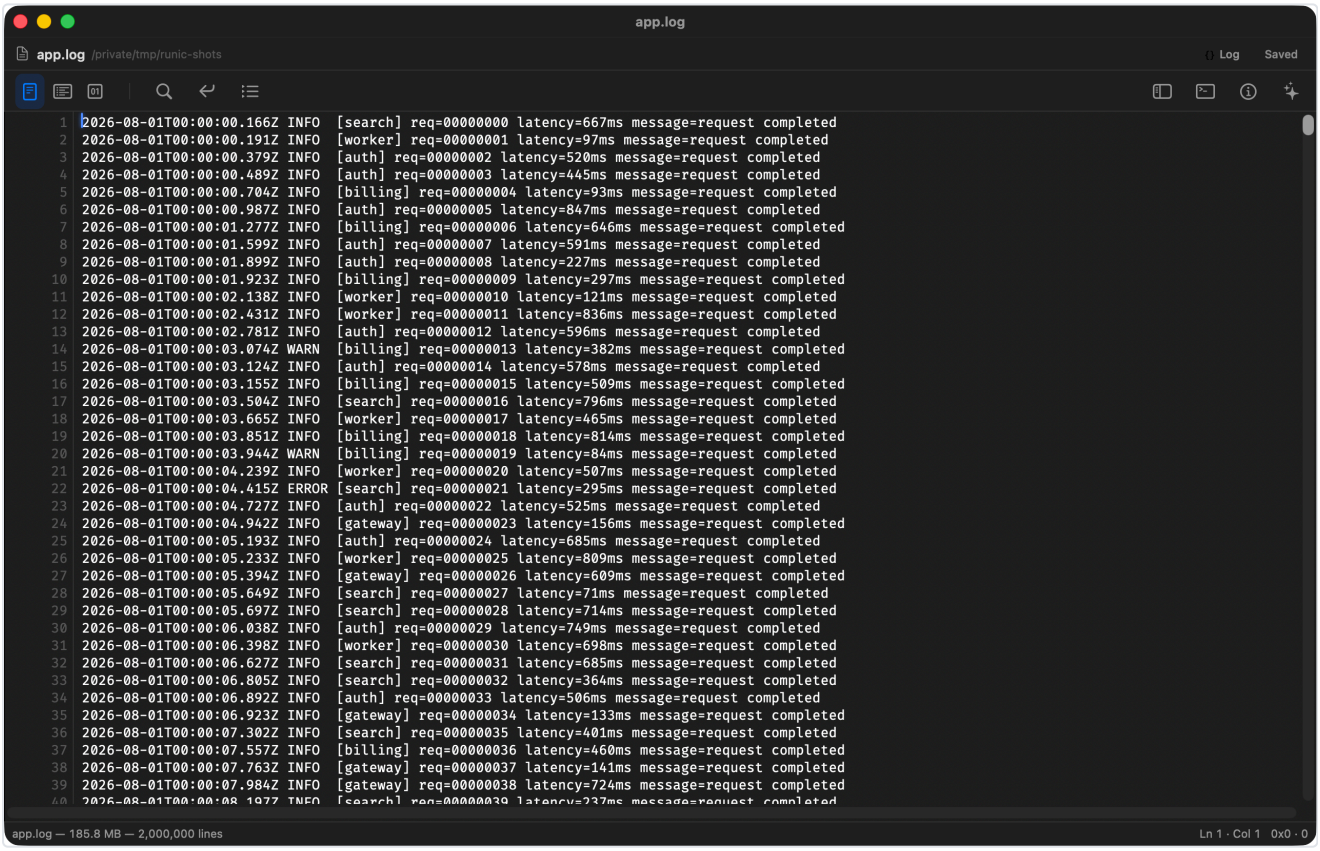
Find bar — collapsed until summoned with `⌘F`.

Ribbon — one row of contextual controls, described below.

Editor — the document, drawn through the current lens.

Status bar — cursor line and column, document size, encoding, and background indexing progress.

The header and status bars are pure display; they never hold controls that exist nowhere else.



The window, with a 186 MB log open in the text lens: header bar, ribbon, gutter and editor, status bar. The status bar reports the full size and line count even though only the visible lines have been read.

The ribbon

The ribbon is a single command strip whose contents depend on three things: the active lens, the document type, and the document's state. It is not a customisable toolbar and it does not accumulate buttons.

Examples of what this means in practice:

- In the **table lens**, the ribbon shows the row filter and **Query Table…**
- In the **log lens**, it shows **Follow**, the log filter, and highlight rules
- In the **hex lens**, it shows number base, byte grouping, and endianness
- In the **text lens** on a JSON file, it offers **Format Document**

Controls in the ribbon are the same commands available in the menus — the ribbon adds no behaviour of its own. A control appears greyed out under exactly the same conditions its menu item does.

Icons are SF Symbols and follow your system appearance. Where the ribbon runs out of room, surplus controls fold into a **More** menu.

Tabs and windows

Runic uses native macOS window tabbing.

ACTION	COMMAND	SHORTCUT
New tab	`window.newTab`	`⌘T`
Next tab	`window.nextTab`	`^Tab`
Previous tab	`window.previousTab`	`^⇧Tab`
Merge all windows	`window.mergeAll`	—
Close	`window.close`	`⌘W`

Each document is an independent window that macOS may group into a tab bar. Commands always act on the frontmost document.

Appearance

Runic follows the system light/dark setting. All interface colours are semantic system colours, so the application changes appearance immediately when macOS does.

The one deliberate exception is the **caret**, which is drawn in Rune Blue with a soft glow — an echo of the lit slit in the application icon.

Menu reference

This section documents every menu and every item. Items that Runic registers as commands carry their command identifier, which you can use to remap the keyboard shortcut (see *Customising keyboard shortcuts*).

Runic menu

ITEM	COMMAND	NOTES
About Runic	`app.about`	Version, build commit, and credits
Settings…	`app.preferences`	Opens the Settings window
Quit Runic	`app.quit`	Prompts if any document has unsaved changes

File menu

ITEM	COMMAND	SHORTCUT
New	`file.new`	⌘N
New Tab	`window.newTab`	⌘T
Open…	`file.open`	⌘O
Open Folder…	`workspace.openFolder`	⌥⇧⌘O
Save	`file.save`	⌘S
Save As…	`file.saveAs`	⇧⌘S
Save In Place (Overwrite)…	`file.saveInPlace`	—
Revert to Saved	`file.revert`	—
Export Selection to File…	`file.exportSelection`	—
Print…	`file.print`	⌥⌘P
Connect to Host…	`remote.connect`	—
Disconnect Host	`remote.disconnect`	—
New Terminal	`terminal.new`	^⌘T
Close	`window.close`	⌘W

New creates an in-memory document with no file behind it.

Open Folder… roots the Navigator panel at a folder. This is what enables Quick Open (`⌘P`) and the folder tree.

Save writes the document atomically: the new contents are streamed to a temporary file in the same directory, flushed to disk, and then swapped into place with an atomic rename. A crash at any point leaves either the complete old file or the complete new one — never a partial write. The original file's permissions, ACLs, and extended attributes are carried across.

Save In Place (Overwrite)… is a deliberate alternative that writes directly over the existing file rather than replacing it. It applies only when your edits preserve the file's length exactly, and it exists for cases where the file's identity must not change — for example, a file that another process holds open, or a device node. Because it is not atomic, it asks for confirmation. Prefer ordinary **Save** unless you specifically need this.

Revert to Saved discards unsaved changes and re-reads the file from disk.

Export Selection to File… writes just the selected bytes to a new file. On a huge file this is the efficient way to carve out a region without loading the whole thing.

Disconnect Host is enabled only for a remote document. It closes the document's window first so that unsaved changes can still be uploaded through the normal save guard while the connection is alive.

Edit menu

The Edit menu holds the standard macOS editing actions. These are deliberately *not* Runic commands: they travel the responder chain so they also work in text fields, search boxes, and dialogs.

ITEM	SHORTCUT
Undo	`⌘Z`
Redo	`⇧⌘Z`
Cut	`⌘X`
Copy	`⌘C`
Paste	`⌘V`
Select All	`⌘A`

Undo history is per document and is not bounded by file size — undoing an edit in a multi-gigabyte file is as fast as making it.

Multiple cursors

ITEM	COMMAND	SHORTCUT
Add Next Occurrence	`selection.addNextOccurrence`	`⌘D`

ITEM	COMMAND	SHORTCUT
Add Cursor Above	`selection.addCaretAbove`	—
Add Cursor Below	`selection.addCaretBelow`	—

Add Next Occurrence finds the next instance of the current selection and adds a cursor there. Repeat to build up a multi-cursor selection, then type once to edit every location.

View menu

ITEM	COMMAND	SHORTCUT
Command Palette…	`command.palette`	⇧⌘P
Quick Open…	`nav.quickOpen`	⌘P
Go to Line / Offset…	`view.goToLine`	⌘L
Go to Byte Offset…	`nav.gotoByte`	^⌘G
Go to Symbol…	`nav.gotoSymbol`	⇧⌘O

Command Palette searches every command by name, including the many that have no menu item or shortcut. If you cannot find a feature, look here first.

Quick Open searches files by name within the folder open in the Navigator. It requires a folder to be open.

Go to Line / Offset… accepts either a line number or a byte offset. **Go to Byte Offset…** always interprets the value as a byte position, which is the one you want in the hex lens.

Go to Symbol… lists the structural symbols in the current file — functions, classes, keys, headings — depending on file type.

View › Show as

Switches the lens. See *Lenses* for what each one does.

ITEM	COMMAND
Text	`view.showAsText`
Hex	`view.showAsBytes`
Table	`view.showAsTable`
Log	`view.showAsLog`
Markdown	`view.markdownReading`

Lens items appear only when they apply to the current document. **Table** is offered for CSV, TSV, and NDJSON; **Markdown** for any small document, including unsaved ones.

View › Panels

Each panel has a toggle, a **Dock** item that switches it between docked and floating, and — for panels that allow more than one position — a **Move … to Other Side** item.

PANEL	TOGGLE COMMAND	SHORTCUT	DEFAULT POSITION
Navigator	`view.outline`	⌥⌘0	Left
Assistant	`ai.toggleChat`	⇧⌘A	Right
Inspector	`view.inspector`	^⌘I	Right
Performance	`view.perf`	—	Right
Shell	`view.shell`	—	Bottom
SQL	—	—	Bottom

Panel positions and visibility are remembered between launches, per panel.

Every panel also has a **Dock …** command (`panel.<name>.toggleDocked`) that switches it between docked and floating. Panels that may sit on either side have **Move … to Other Side** (`panel.<name>.toggleSide`). The Shell and SQL panels dock at the bottom only, so they have a Dock command but no Move command.

The SQL panel has no toggle of its own — it appears when you run **Query Table…** from the table lens.

View — display toggles

Each of these is a checkmark item: the menu shows a tick when it is on, and the setting is remembered.

ITEM	COMMAND	SHORTCUT	EFFECT
Wrap Lines	`view.toggleWrap`	⌥Z	Soft-wrap long lines instead of scrolling horizontally
Minimap	`view.toggleMinimap`	—	A condensed overview of the document beside the scrollbar
Sticky Scroll	`view.toggleSticky`	—	Pin the enclosing context to the top of the editor while scrolling
Rainbow Brackets	`view.toggleRainbow`	—	Colour nested brackets by depth

ITEM	COMMAND	SHORTCUT	EFFECT
Value Decorations	`view.toggleDecorations`	—	Inline annotations on values
Outline Mode	`view.toggleOutlineMode`	—	Treat indentation as structure, giving an outline for files with no grammar
Change Bars	`view.toggleChangeBars`	—	Mark lines edited since the last save in the gutter
Ribbon	`view.toggleRibbon`	—	Show or hide the contextual control strip

Outline Mode is worth knowing about: it derives an outline from indentation, so plain text, notes, and configuration files without a tree-sitter grammar still get a navigable structure in the Navigator's Outline section.

Change Bars and **Show Changes…** work together. Change bars mark which lines you have edited since the last save; **Show Changes…** (`view.showChanges`) opens a review of those unsaved edits against the file on disk, and lets you revert individual hunks. It is the fastest way to answer "what have I actually changed?" before saving.

View — workspace

ITEM	COMMAND	EFFECT
Reveal in Sidebar	`workspace.revealInTree`	Expand the Navigator's folder tree to the current document and select it
Show Hidden Files in Sidebar	`workspace.toggleHidden`	Show dotfiles in the folder tree

Code menu

ITEM	COMMAND	SHORTCUT
Fold / Unfold	`fold.toggle`	`⌥⌘.`
Fold All	`fold.all`	—
Unfold All	`fold.unfoldAll`	—
Format Document	`format.document`	`⌥⇧F`
Minify Document	`format.minify`	—
Trim Trailing Whitespace	`transform.trimTrailing`	—

ITEM	COMMAND	SHORTCUT
Collapse Whitespace	`transform.collapseWhitespace`	—
Join Lines	`transform.joinLines`	—
Sort Selected Lines	`lines.sort`	—
Remove Duplicate Lines	`lines.dedup`	—
Filter Through Command…	`transform.filterCommand`	—
Add Highlight Rule…	`decorations.addRule`	—
Clear Highlight Rules	`decorations.clearRules`	—
Reopen with Encoding…	`encoding.reopen`	—
Convert to UTF-8	`encoding.convertToUTF8`	—
Validate Against JSON Schema…	`schema.validate`	—

These are documented in detail under *Reshaping documents*, *Encodings*, and *Highlight rules*.

AI menu

ITEM	COMMAND
Edit Selection with AI…	`ai.editSelection`
AI Key Status	`ai.keyStatus`
Choose AI Provider…	`ai.chooseProvider`
Set API Key…	`ai.setApiKey`

AI Key Status is informational — it shows the selected provider and whether its key is stored and verified. See *The AI assistant*.

Window menu

Standard macOS window commands — Minimize, Zoom, and the window list — plus the tab commands listed under *Tabs and windows*.

Help menu

ITEM	COMMAND
Check for Updates…	`help.checkForUpdates`
Acknowledgements	`help.acknowledgements`

ITEM	COMMAND
Privacy Policy	`help.privacy`

Acknowledgements lists the licences of third-party components. **Privacy Policy** describes what Runic stores locally and what — if anything — leaves your machine.

Lenses

A lens changes how Runic draws a document. It does not convert the file, create a second document, or alter the bytes. Switching lenses is instantaneous and lossless, and the text lens is always available.

When you open a file, Runic suggests a lens based on the file extension and a sample of the content. A binary file opens in the hex lens; a `.csv` opens as a table. The suggestion never locks you in.

Text lens

The default. Standard editing with syntax highlighting, folding, bracket colouring, sticky scroll, and the structural breadcrumb.

Syntax highlighting is driven by tree-sitter grammars. Runic ships grammars for JSON, Python, JavaScript, Rust, C, C++, YAML, TOML, Go, Bash, HTML, CSS, Swift, and XML. Grammars are loaded on demand the first time you open a file of that type, so the application starts quickly regardless of how many are installed. HTML documents additionally highlight embedded CSS and JavaScript.

Highlighting is computed for the visible region only. A file with no matching grammar simply displays as plain text.

Folding collapses structural regions. `⌥⌘.` folds or unfolds the region containing the cursor; **Fold All** and **Unfold All** apply to the whole document.

Sticky scroll keeps the enclosing context — the object key or function signature you are inside — pinned at the top of the editor as you scroll within it.

Rainbow brackets colour nested brackets by depth so matching pairs are visually obvious.

Soft wrapping

Wrap Lines (`⌥Z`) wraps long lines to the window width instead of scrolling horizontally.

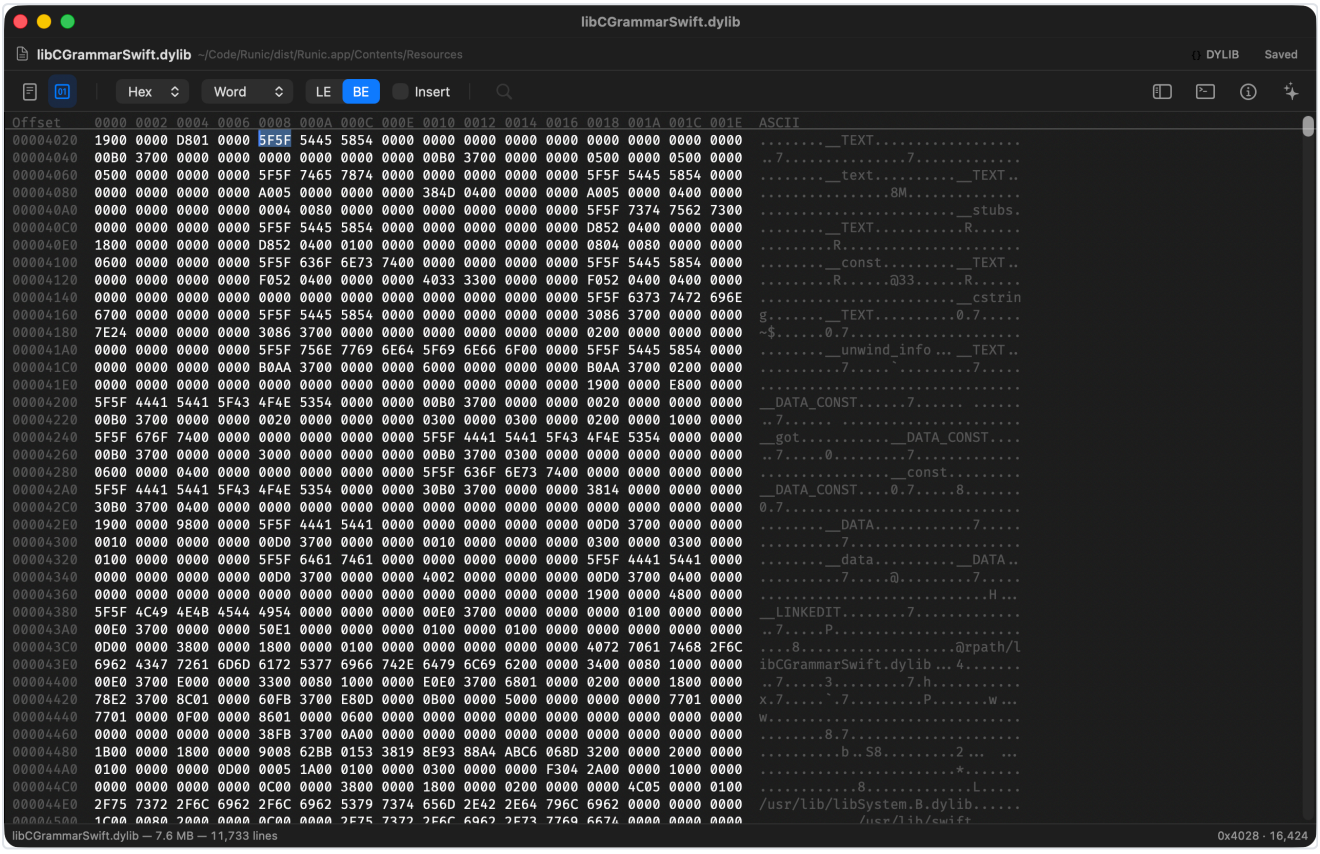
Wrapping is computed only for the lines on screen, which is what keeps it fast on a large file. Two consequences are worth knowing:

- The vertical scrollbar remains based on logical lines, so it is approximate when wrapping is on. This is a deliberate trade for wrap being instant and stable when you resize the window.
- A single line longer than 64 KB — a minified JSON file, typically — is **not** wrapped even when wrapping is on, because laying out millions of visual rows would be slow. Runic shows a banner offering to pretty-print it instead, which is the better fix.

Hex lens

Displays the file as `offset | bytes | ASCII`. Use it for binary files, for inspecting exact byte values, or for any file where encoding is in question.

Editing in the hex lens edits raw bytes. Bytes that are not valid UTF-8 survive exactly, and the text and hex views of the same document stay in sync.



The hex lens on a Mach-O dylib, positioned at the segment load commands. The ribbon carries the radix (hex, decimal, octal, binary), the grouping, the byte order, and the Insert toggle.

Ribbon controls:

CONTROL	COMMANDS	OPTIONS
Number base	`view.radix.hexadecimal`, `.decimal`, `.octal`, `.binary`	Hex, decimal, octal, binary
Grouping	`view.byteUnit.1`, `.2`, `.4`, `.8`	Byte, word, dword, qword
Byte order	`view.byteEndian.little`, `.big`	Little-endian, big-endian
Insert mode	`bytes.toggleInsertMode`	Overwrite (default) or insert

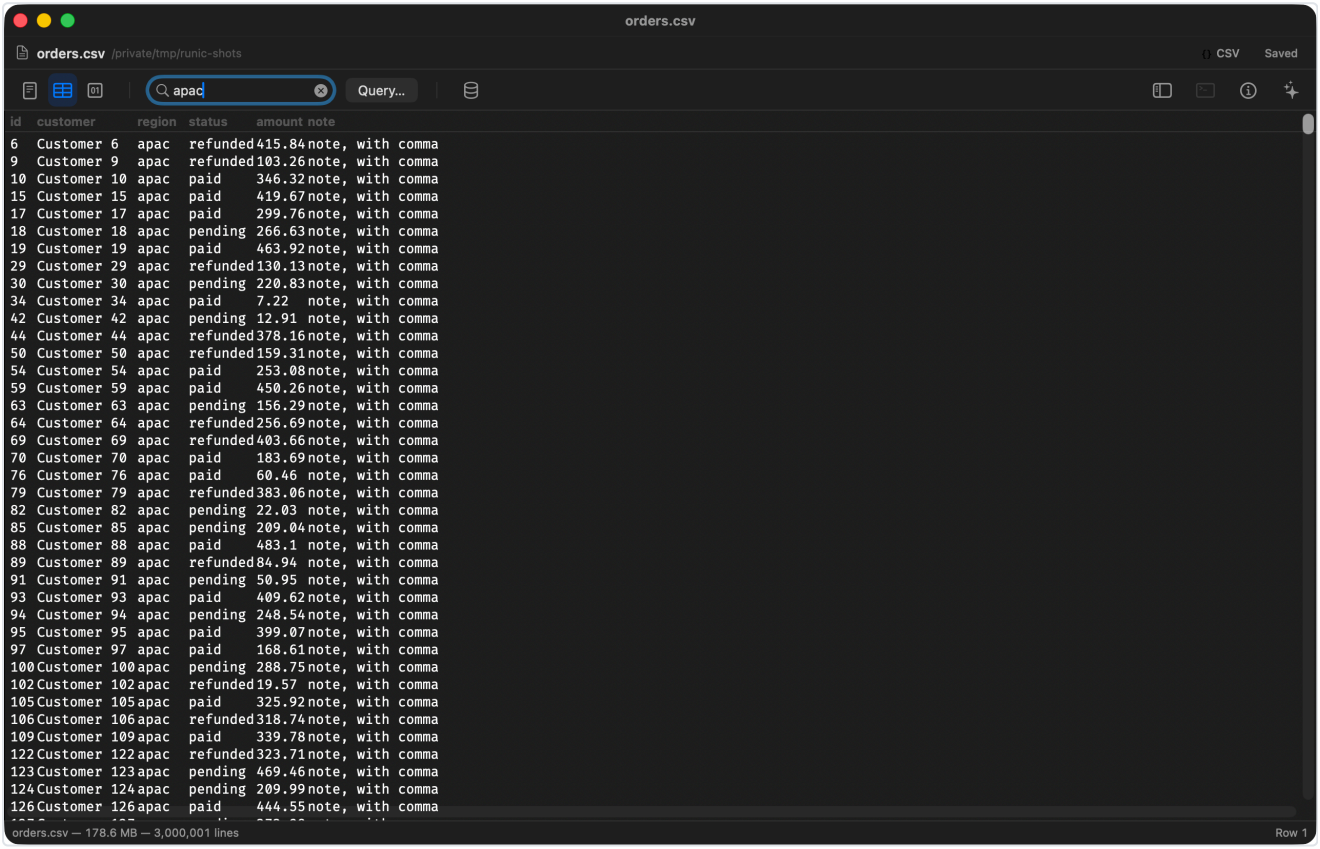
Byte Insert Mode changes whether typing replaces the byte under the cursor or inserts a new one, shifting the rest of the file. Overwrite is the default because it is the safe behaviour for fixed-layout binary formats.

Grouping and byte order affect display only; they never change the file.

Table lens

Renders CSV, TSV, and NDJSON as rows and columns. Rows and columns are both virtualized, so file size and column count are not limits.

CSV and TSV are parsed with full quoting rules: a newline inside a quoted field stays part of one record, and quoted separators are not treated as delimiters. This is correct from the first row rather than being a fast-path approximation.



The table lens on a 179 MB CSV of three million rows, filtered to `apac`. The `note` column holds a quoted field containing a comma, parsed as one value rather than split.

NDJSON files are indexed by line, with column names discovered lazily from the records as you scroll.

Ribbon controls:

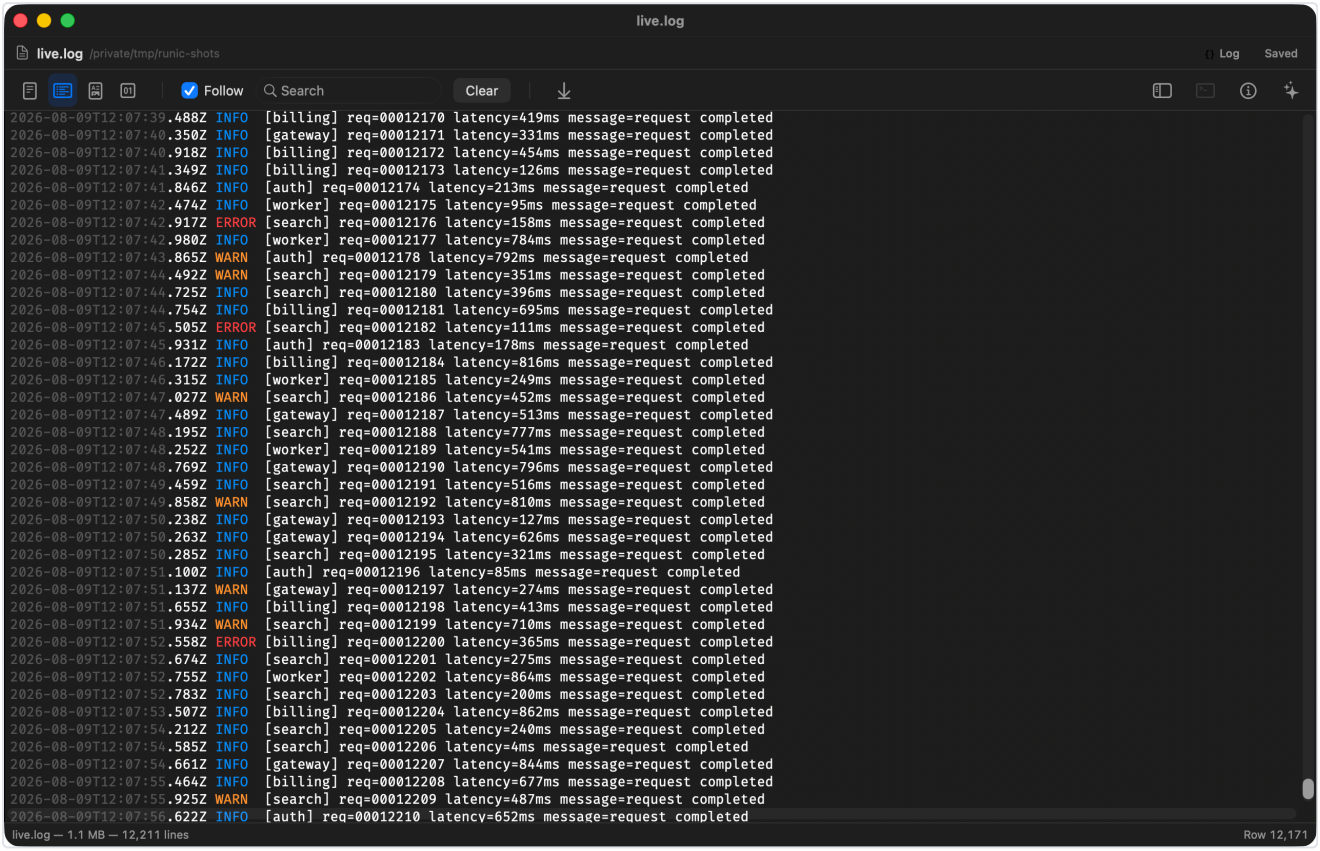
- **Filter Table…** (`table.filter`) — narrows to rows containing the text you type, streaming through the file in the background with a live match count. Large files remain responsive while it scans.
- **Query Table…** (`table.query`) — runs SQL against the file. See *Querying data with SQL*.

Clicking a row pretty-prints that record, which is the practical way to read a single dense NDJSON line.

Log lens

The text lens plus three things built for logs.

Follow (Tail) (`log.toggleFollow`) sticks the view to the end of the file as it grows, like `tail -f` . Combined with automatic reloading of unmodified files, this makes Runic a live log viewer.



The log lens following a file that is still being appended to. The view stays pinned to the newest line, and the default highlight rules colour `ERROR` and `WARN` .

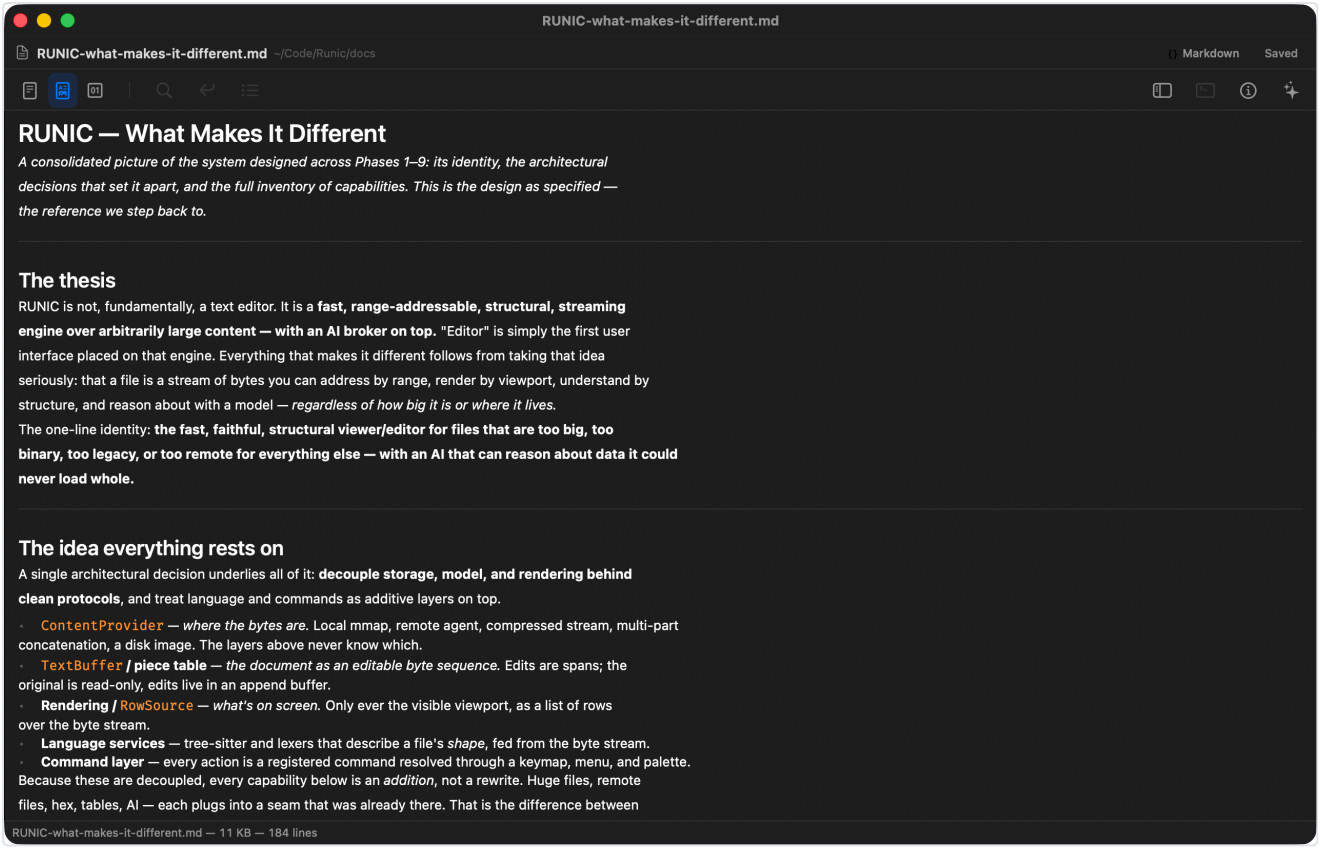
Filter Log… (`log.setFilter`) shows only matching lines, streaming as it scans, with a live count. **Clear Log Filter** (`log.clearFilter`) restores the full view.

Highlight rules colour lines matching a regular expression — see below. They apply to the visible region only and are independent of syntax highlighting.

Markdown reading lens

Renders Markdown rather than showing its source: headings, emphasis, lists, tables, code blocks, links, and images.

The renderer is native — Runic does not embed a web browser. Raw HTML in a Markdown document is escaped and shown as text rather than rendered, and mathematical notation and diagram languages are not supported.



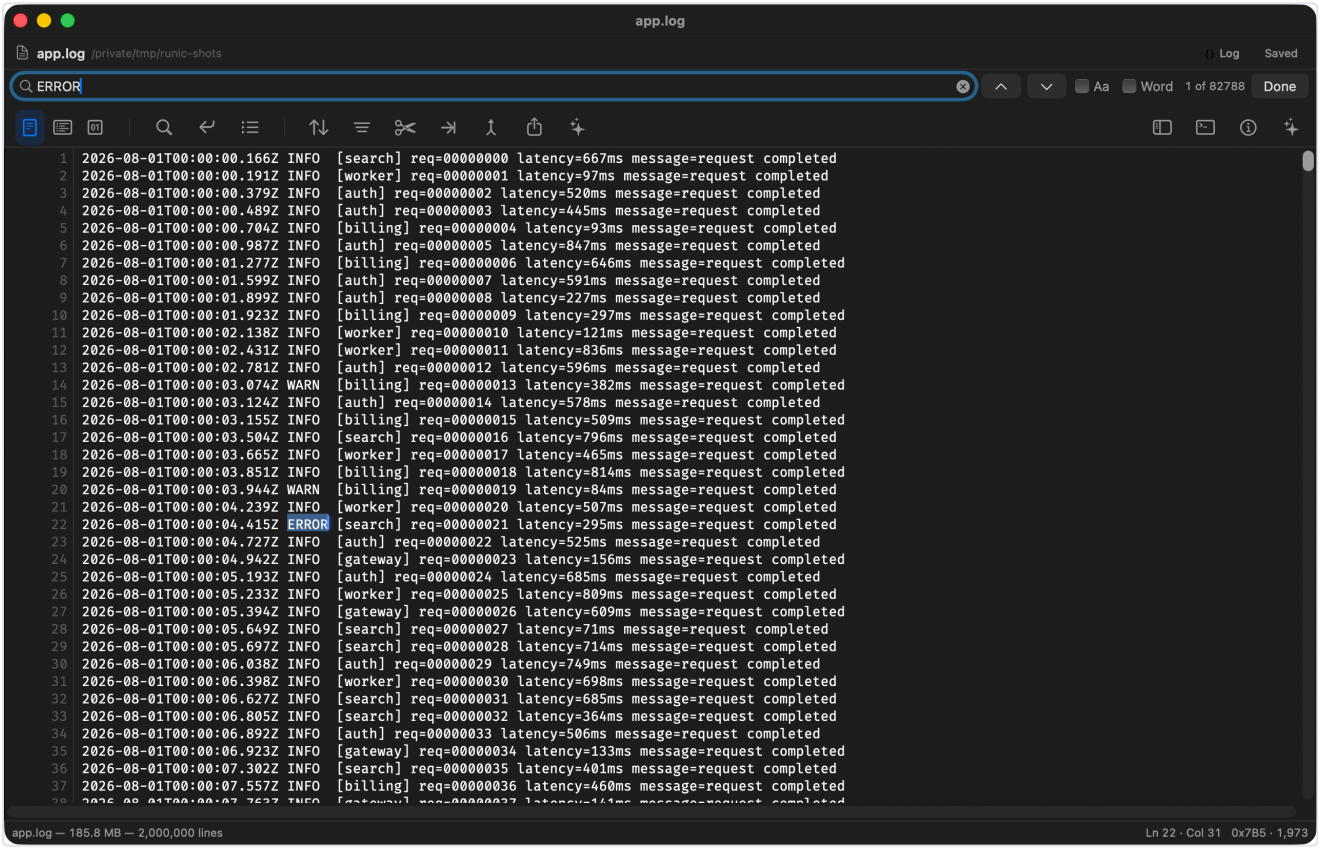
The Markdown reading lens. Headings, emphasis, lists, rules, and inline code are drawn by the app itself.

This lens is offered for any small document, including unsaved ones, not only for saved `.md` files.

Finding and replacing

Press `⌘F` to open the find bar.

ACTION	COMMAND	SHORTCUT
Find…	`find.show`	`⌘F`
Find and Replace…	`replace.toggle`	`⌥⌘F`
Find Next	`find.next`	`⌘G`
Find Previous	`find.previous`	`⇧⌘G`
Close Find	`find.close`	`Esc`



The find bar on a two-million-line log. The match count (82,788) is reported as the stream is scanned, so the first matches are usable before the scan finishes.

Search streams over the document rather than loading it, so results begin appearing immediately on a file of any size, and matches spanning internal chunk boundaries are found exactly once.

Options in the find bar include case sensitivity and whole-word matching. Replace operates on the current match or on all matches.

For a remote document with the host agent installed, **search runs on the server** and only the match positions are sent back. Searching a 50 GB remote file does not transfer the file.

Highlight rules

Code › Add Highlight Rule… (`decorations.addRule`) colours any text matching a regular expression. Rules are saved and apply to every document you open, which makes them useful for things you always want to spot: `ERROR` in red, a customer id in blue, a deprecated API name in amber.

Clear Highlight Rules (`decorations.clearRules`) removes all of them.

Highlight rules are evaluated for visible lines only, and are independent of syntax highlighting — they work in files that have no grammar at all, which is the usual case for logs.

Reshaping documents

Formatting

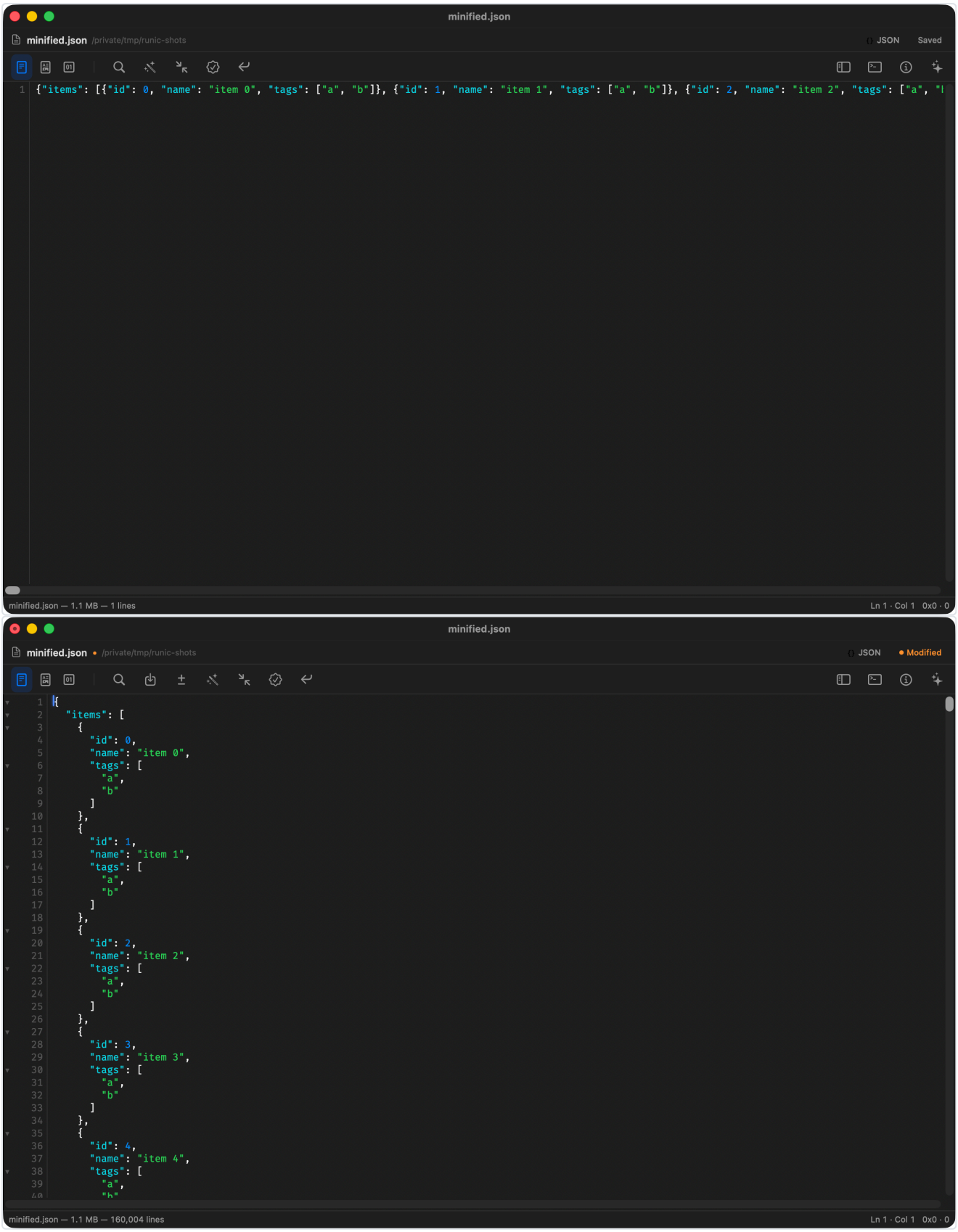
Format Document (`⌥⇧F` , `format.document`) pretty-prints JSON, XML, and TOML. **Minify Document** (`format.minify`) compacts JSON and XML.

Formatters stream: they re-emit the document with adjusted whitespace while reading it in chunks, using memory proportional to nesting depth rather than file size. You can pretty-print a file larger than available RAM.

What is preserved exactly:

- **JSON** — string contents, key order, and the original spelling of numbers
- **XML** — mixed content, anything inside `xml:space="preserve"` , comments, CDATA, processing instructions, and the DOCTYPE, all byte-identical
- **TOML** — comments, ordering, every string form, and values spanning multiple lines

How the edit is applied depends on size. Below 16 MB, the whole document is replaced as a single undoable change. Above that, the result is streamed to a new file, which is not undoable in place.



Format Document on a 1.1 MB minified JSON file — before (top) and after (bottom). One line becomes 160,004, indented and foldable, as a single undoable step.

> **The JSON formatter is a re-indenter, not a validator.** It checks that brackets and braces balance and that separators are present, and reports the byte offset of a structural error. It will accept some input that is balanced but not strictly valid JSON. To actually validate a document, use **Validate Against JSON Schema…**.

Text transforms

Under the **Code** menu:

COMMAND	EFFECT
Trim Trailing Whitespace	Removes spaces and tabs at the end of every line
Collapse Whitespace	Collapses runs of spaces and tabs to a single space. Confirms first.
Join Lines	Removes all line breaks. Confirms first.
Sort Selected Lines	Sorts the selected lines A→Z, case-insensitively
Remove Duplicate Lines	Drops repeated lines within the selection, keeping the first

Sort Selected Lines and **Remove Duplicate Lines** require a selection. **Collapse Whitespace** and **Join Lines** ask for confirmation because they change the entire document irreversibly in one step.

Filter Through Command

Filter Through Command… (`transform.filterCommand`) pipes the selection — or the whole document if nothing is selected — through a shell command, and replaces it with the output.

This turns any command-line tool into an editor command:

```
sort -u
jq '.items[] | .name'
sed 's/foo/bar/g'
tr 'a-z' 'A-Z'
```

The replacement is a single undoable change, so `⌘Z` restores the original if the result is not what you wanted.

Validating JSON

Validate Against JSON Schema… (`schema.validate`) checks the open JSON document against a JSON Schema file you choose, and reports failures with their locations. The supported subset is JSON

Schema Draft 7.

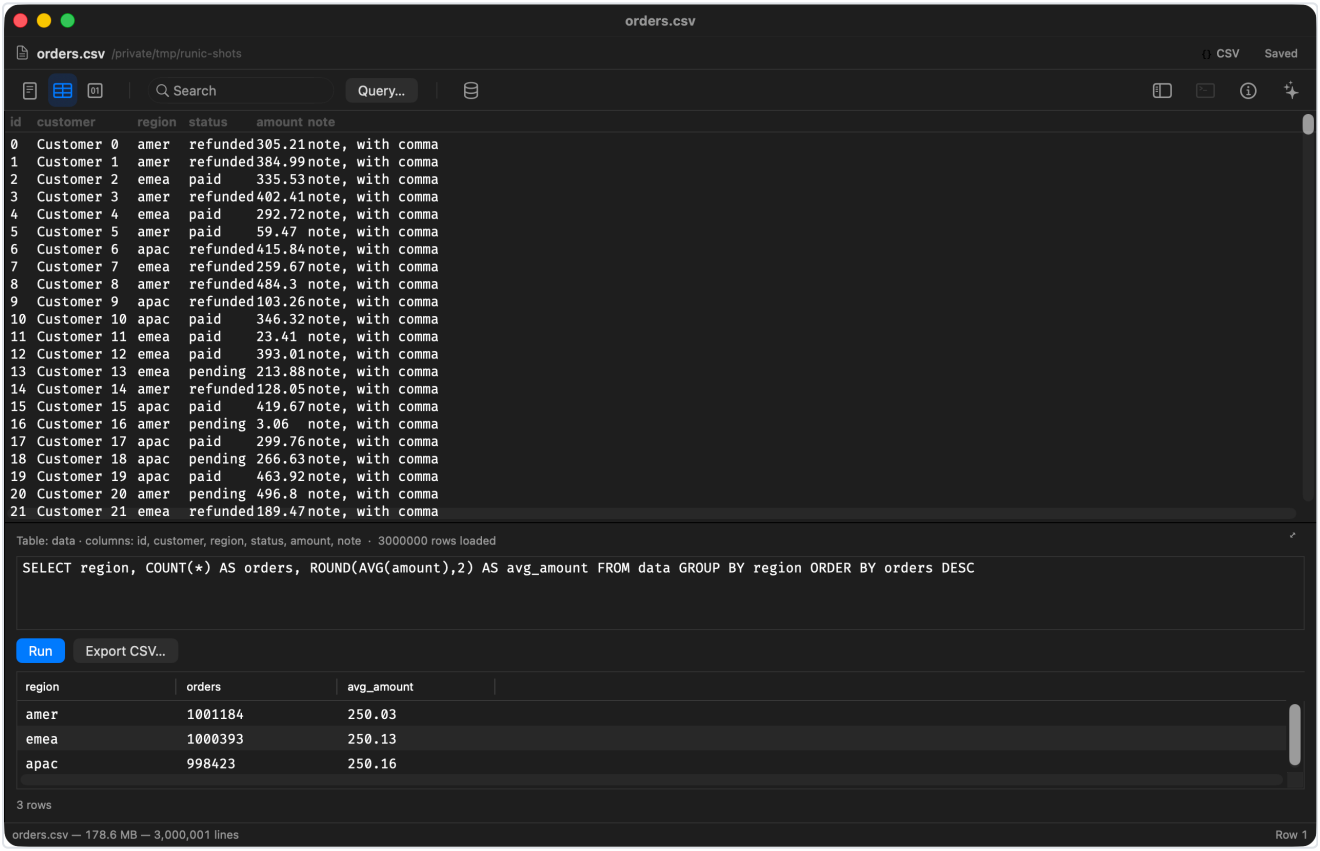
Querying data with SQL

With a CSV, TSV, or NDJSON file open in the table lens, **Query Table…** (`table.query`) runs SQL against it. The table is named `data`:

```
SELECT status, COUNT(*) AS n
FROM data
GROUP BY status
ORDER BY n DESC
```

Results appear in the **SQL** panel at the bottom of the window.

The file is attached as a read-only virtual table and cells are read on demand directly from the mapped file. Nothing is imported, copied, or materialised beyond the result set — so there is no import step, no row limit, and no memory cost proportional to file size.



The SQL panel, docked below the table lens, aggregating three million rows. The query reads through the virtual table straight off the mapped file.

Numeric comparisons work as you would expect (`WHERE amount > 100`) despite the source being text.

The SQL panel binds to the active table document, rebinding when you switch windows and showing an empty state when the frontmost document is not a table. Querying never modifies the document.

Panels

Panels dock to the left, right, or bottom of the window, or float in their own window. Toggle them under **View › Panels**, float a docked panel with the ⤢ button in its header, and return a floating panel to the window using the blue button in its title bar.

Each panel remembers whether it is visible, which side it is on, and whether it is docked or floating, across launches.

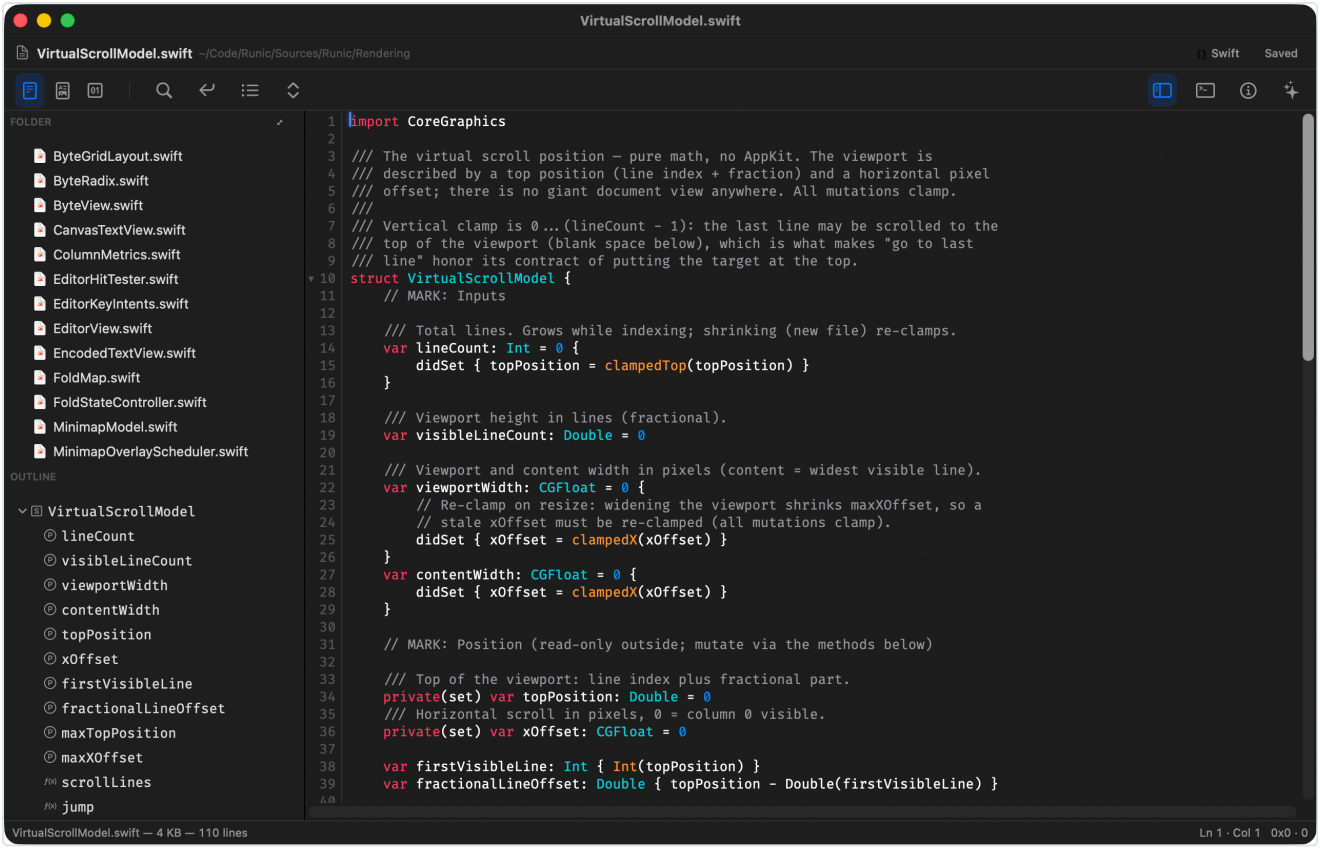
Docked panels follow the active window: bringing a different document to the front re-hosts the panels there rather than leaving them behind.

Navigator

`⌥⌘0` — two collapsible sections in one panel.

Folder shows the folder tree opened with **File › Open Folder…**. It loads lazily as you expand, so a large repository costs nothing up front. Dotfiles and anything matched by `.gitignore` are hidden by default. Clicking a file opens it in the existing tabs.

Outline shows the structure of the document you are editing — headings in Markdown, keys in JSON, declarations in source code — derived from the same parse tree that drives highlighting. Clicking an entry jumps to it.



The Navigator on a Swift file, both sections open: the **Folder** tree shares the panel with the **Outline**, whose symbols come from the parse tree. Collapsing one section gives the whole panel to the other.

The Navigator can be rooted at a **remote** directory as well as a local one; see *Remote files over SSH*.

Assistant

`⇧⌘A` — the AI chat panel. See *The AI assistant*.

Inspector

`^⌘I` — details about the current document: size, encoding, line count, and related state.

SQL

Query results for the table lens. See *Querying data with SQL*.

Shell

A terminal docked at the bottom of the window. **File › New Terminal** (`^⌘T`) opens a terminal as well. The shell used is configurable in Settings.

Performance

Live timings from the rendering and indexing engine. This panel is diagnostic — useful when investigating whether a file or a feature is behaving unusually, and unnecessary otherwise.

Encodings

Runic detects a file's text encoding when opening it, in this order: a byte-order mark if present; a NUL-density test that identifies UTF-16; a UTF-8 validity check; and finally Windows-1252 as a fallback for legacy single-byte text.

UTF-8 is the native format — it is what the editor reads and writes.

Reopen with Encoding… (`encoding.reopen`) re-decodes the file with an encoding you choose, for the cases detection cannot resolve on its own.

Convert to UTF-8 (`encoding.convertToUTF8`) transcodes a UTF-16, UTF-32, or legacy file into UTF-8. This is an explicit transformation, and it is what makes such a file fully editable — files in other encodings open read-only, decoding only the visible rows.

Throughout, Runic indexes lines on the `\n` character and strips a trailing `\r` at display time, so files with Windows line endings display correctly.

Large files

Everything below happens automatically. This section explains what you are seeing.

Opening. A file of any size paints its first screen in well under a second. Runic maps the file and reads only what it draws.

Indexing. Finding line boundaries happens on a background thread. The status bar shows progress. The indexed portion is fully usable — scrollable and searchable — while the rest continues, so you never wait for indexing to finish.

Scrolling. Scrolling is virtual: Runic draws a fixed-size canvas and moves through the file, rather than building a document view as tall as the file. This is why scroll position stays precise in files where a conventional editor's scrollbar becomes unusable.

Editing. Edits are recorded as a set of changes over the original bytes rather than by rewriting the file, so inserting text in the middle of a huge file is instant, as is undo.

Saving. Saving streams the result to a new file and swaps it in atomically.

Compressed files. A `.gz` file opens decompressed automatically.

Remote files over SSH

Runic edits files on other machines over SSH. Rendering, search, highlighting, and editing are identical to local files, because every one of them only ever asks for the bytes currently needed.

Connecting

File › Connect to Host… (`remote.connect`) asks for a host and a path.

Enter the host as `user@host` — for example `root@virtual`. Hosts defined in your `~/.ssh/config` appear in the drop-down. A bare `host` tells ssh to log in with your *local* username, which is usually not what you want on a server.

What happens next depends on what the path points at:

- **A file** opens in a new tab.
- **A directory** opens in the Navigator panel, where you can browse it and open files from it.

Authentication

Runic uses your own `ssh` and stores no credentials of any kind. Keys, `~/.ssh/config`, `known_hosts`, ssh-agent, jump hosts, and multi-factor prompts are all handled by ssh exactly as they are in Terminal. **Runic never asks for, transmits, or stores a password.**

One consequence: because Runic runs as a graphical application, ssh has no terminal on which to show an interactive password prompt. **Key-based authentication is required.** If a host accepts only passwords, connect to it once in Terminal first, or configure a key.

If `ssh user@host` works in Terminal, Runic will work.

The host agent

Runic works over plain ssh, but installing the **host agent** on the server changes what is possible.

The agent is a small headless program — no interface, never launched by hand. Runic starts it over ssh when you connect. With it running on the server:

- Line indexing happens on the server
- Search runs on the server; only match positions cross the network
- Saving uploads only the parts of the file that changed
- Directories can be browsed in the Navigator

Without it, Runic falls back to reading the file over plain ssh. Everything still works, but indexing and search run locally, which means streaming the file to your Mac. The header shows · **no agent** whenever

you are in this reduced mode, so you always know.

To install it, place the agent binary on the host's `PATH` as `runic`:

```
scp runic-agent user@host:/tmp/runic
ssh user@host 'sudo install -m 0755 /tmp/runic /usr/local/bin/runic'
```

Verify with `ssh user@host runic --version`.

Saving remote files

Saving sends only what changed: the parts of the file already on the server are referenced by position, and only the bytes you actually typed are uploaded. The server reassembles the file and swaps it into place atomically, with the same crash-safety guarantees as a local save. Editing a few lines of a very large remote file uploads kilobytes.

Connection loss

If the connection drops, your unsaved edits are held locally and are not lost. The document is marked offline and Runic attempts to reconnect.

Disconnecting

File › Disconnect Host (`remote.disconnect`) closes the current remote document and its connection. The window closes first, so a document with unsaved changes still goes through the normal save prompt while the connection is live.

The AI assistant

Runic integrates an AI assistant for editing and answering questions about the document. Three properties are structural rather than optional, and are worth understanding before you use it.

What gets sent

Never the whole file. Runic assembles a bounded context from prioritised sources: your selection first, then the enclosing structural node (the function, object, or block your cursor is in), then the visible region, then project metadata. The total is capped by construction.

Before anything is sent, the assembled context is shown as removable chips with a byte count, so you can see and reduce exactly what will leave your machine. A cursor inside a function in a 2 GB file sends that function.

Where your key lives

Your API key is stored in the macOS Keychain. It is never written to a configuration file, never included in a URL, and never logged. Document content travels in the request body; the URL is only the endpoint.

How edits are applied

An AI response that proposes changes becomes a **diff you review**. You accept or reject each hunk individually. Applying the accepted hunks is a single undo step, so `⌘Z` reverses the entire AI edit at once.

AI edits never bypass this review.

Providers

AI › Choose AI Provider… (`ai.chooseProvider`) selects between:

- **Anthropic** — Claude models
- **OpenAI-compatible** — any endpoint implementing that API
- **Ollama** — models running locally on your own machine, in which case nothing leaves it at all

Each provider has its own stored key and model setting. **AI › Set API Key…** (`ai.setApiKey`) stores the key for the selected provider; **AI › AI Key Status** (`ai.keyStatus`) reports which provider is active and whether its key is present and verified.

Using it

AI › Edit Selection with AI… (`ai.editSelection`) rewrites a selection according to an instruction, returning a reviewable diff.

The **Assistant** panel (`⇧⌘A`) is a conversation about the current document, with responses streaming in and rendered as Markdown.

Settings

Runic › Settings… (`app.preferences`).

SETTING	KEY	NOTES
Font	`settings.fontName`, `settings.fontSize`	Monospaced fonts only — the renderer depends on a fixed character cell
Tab width	`settings.tabWidth`	Columns per tab stop
Insert spaces	`settings.insertSpaces`	Whether Tab inserts spaces
Theme	`settings.themePreset`	Follows the system light/dark setting
Outline glyphs	`settings.outlineGlyphs`	Symbol icons in the outline
Byte radix	`settings.byteRadix`	Default number base in the hex lens
Byte grouping	`settings.byteUnit`	Default grouping in the hex lens
Byte order	`settings.byteEndianness`	Default endianness in the hex lens
Terminal type	`settings.terminalType`	Shell used by the Shell panel
Auto-reload clean documents	`autoReloadCleanDocuments`	On by default; see below

Auto-reload clean documents controls what happens when a file changes on disk while open. A document you have *not* edited reloads automatically — the behaviour that makes Runic a live log viewer. A document you *have* edited never reloads silently; it shows a conflict banner, and your edits are preserved regardless of this setting.

Customising keyboard shortcuts

Every command in Runic can be bound to a different key. Shortcuts resolve in three layers, each overriding the one before: the built-in defaults, the bundled default keymap, and finally your own file at:

```
~/Library/Application Support/Runic/keymap.json
```

The file maps command identifiers to chords:

```
{
  "format.document": "cmd+shift+l",
  "view.toggleWrap": "ctrl+w"
}
```

Modifiers are `cmd`, `shift`, `alt` (Option), and `ctrl`, joined with `+`.

Each command has exactly one binding: binding a command to a new chord releases its previous one, so a remapping moves the shortcut rather than creating a second live binding.

The complete list of command identifiers is in *Command reference*.

Files Runic creates

PATH	CONTENTS
`~/Library/Application Support/Runic/keymap.json`	Your keyboard shortcut overrides
`~/Library/Application Support/Runic/theme.json`	Theme customisation
macOS Keychain	AI provider API keys
macOS user defaults	Window state, panel layout, recent files, settings

Troubleshooting

A remote connection fails. Check that `ssh user@host` works in Terminal. Remember that Runic requires key-based authentication — it has no terminal on which to answer a password prompt. If you entered a bare hostname, try `user@host` instead.

A remote host shows "· no agent". The host agent is not installed or is not on the remote `PATH`. Runic still works in reduced mode. Verify with `ssh user@host runic --version`.

Folders will not open on a remote host. Directory browsing requires the host agent; the plain-ssh fallback cannot enumerate directories. Runic tells you this rather than failing silently.

A large single line will not wrap. Lines over 64 KB are intentionally not wrapped. Use **Format Document** to break the content across real lines.

Syntax highlighting is missing. Runic ships grammars for a fixed set of languages; a file of another type displays as plain text. Highlighting also falls back to plain text rather than failing if a grammar cannot be loaded.

A file opened as hex unexpectedly. Runic sampled the beginning of the file and found bytes that suggest binary content. Switch with **View › Show as › Text**.

The document reloaded on its own. That is auto-reload for unmodified files. It never happens to a document with unsaved changes. Turn it off in Settings if it is not what you want.

Checking exactly which build you are running. Runic › About Runic shows the version and source commit. From a terminal:

```
/Applications/Runic.app/Contents/MacOS/Runic --version
```

Command reference

Every command Runic registers, grouped by where it appears. The **Command** column is the identifier used in `keymap.json`.

Commands with no menu location are reachable through the command palette (⇧⌘P), a keyboard shortcut, or a ribbon control.

File

ITEM	COMMAND	SHORTCUT	DESCRIPTION
New	`file.new`	⌘N	
New Tab	`window.newTab`	⌘T	
New Terminal	`terminal.new`	^⌘T	Open an interactive local shell in a new tab
Open…	`file.open`	⌘O	
Open Folder…	`workspace.openFolder`	⌥⇧⌘O	Open a folder in the navigator sidebar
Connect to Host…	`remote.connect`		Open a file on a remote host over ssh
Disconnect Host	`remote.disconnect`		Close the remote connection (current doc)
Save	`file.save`	⌘S	
Save As…	`file.saveAs`	⇧⌘S	
Lock Document ✓	`file.toggleLock`	^⌘L	Block edits, saving and reloads for this window
Save In Place (Overwrite)…	`file.saveInPlace`		Write changed bytes directly — non-atomic, for huge/binary files
Reload from Disk	`file.reloadFromDisk`		Load the file's current contents, discarding what's shown
Revert to Saved	`file.revert`		
Print…	`file.print`	⌥⌘P	
Export Selection to File…	`file.exportSelection`		Carve the selected bytes to a new file (streamed)

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Close	`window.close`	`⌘W`	

View

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Go to Line / Offset…	`view.goToLine`	`⌘L`	Jump to a line, byte offset, or row (by view)
Go to Byte Offset…	`nav.gotoByte`	`^⌘G`	Jump to an absolute byte offset (text caret / hex / log) — decimal or 0x…
Quick Open…	`nav.quickOpen`	`⌘P`	Fuzzy-find a file in the workspace folder and open it
Go to Symbol…	`nav.gotoSymbol`	`⇧⌘O`	Fuzzy-find a symbol in the current file's outline and jump to it
Reveal in Sidebar	`workspace.revealInTree`		Select the current file in the folder tree
Command Palette…	`command.palette`	`⇧⌘P`	
Wrap Lines ✓	`view.toggleWrap`	`⌥Z`	Soft-wrap long lines (text view)
Value Decorations ✓	`view.toggleDecorations`		Color swatches, decoded timestamps, URL/UUID/IP underlines (text view)
Outline Mode ✓	`view.toggleOutlineMode`		Indent = hierarchy; Tab/Enter outline gestures (plain text)
Rainbow Brackets ✓	`view.toggleRainbow`		Color brackets by nesting depth (text view)
Sticky Scroll ✓	`view.toggleSticky`		Pin enclosing container headers while scrolling (text view)
Minimap ✓	`view.toggleMinimap`	`⌥⌘M`	Sampled code-shape overview strip on the right (text view)

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Ribbon ✓	`view.toggleRibbon`		Show the contextual action ribbon above the editor
Change Bars ✓	`view.toggleChangeBars`		Mark lines edited since the last save in the gutter (text view)
Show Changes…	`view.showChanges`		Review unsaved edits vs the saved file (revert per hunk)
Show Hidden Files in Sidebar ✓	`workspace.toggleHidden`		Toggle dotfiles in the sidebar

View › Panels

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Navigator ✓	`view.outline`	⌥⌘0	Folder tree + document outline
Assistant ✓	`ai.toggleChat`	⇧⌘A	Open the AI chat panel (needs an API key)
Inspector ✓	`view.inspector`	^⌘I	Show the read-only File Inspector (size, encoding, storage, save strategy)
Performance ✓	`view.perf`		Show the Performance dashboard (memory, indexing, timing)
Shell ✓	`view.shell`		Run a command; insert its output at the cursor
Move Navigator to Other Side	`panel.navigator.toggleSide`		Dock the navigator on the left or right (persisted)
Move Assistant to Other Side	`panel.assistant.toggleSide`		Dock the assistant on the left or right (persisted)
Move Inspector to Other Side	`panel.inspector.toggleSide`		Dock the inspector on the left or right (persisted)

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Move Performance to Other Side	`panel.perf.toggleSide`		Dock the performance on the left or right (persisted)
Dock Navigator	`panel.navigator.toggleDocked`		Dock the navigator panel in the window or float it
Dock Assistant	`panel.assistant.toggleDocked`		Dock the assistant panel in the window or float it
Dock Inspector	`panel.inspector.toggleDocked`		Dock the inspector panel in the window or float it
Dock Performance	`panel.perf.toggleDocked`		Dock the performance panel in the window or float it
Dock SQL	`panel.sql.toggleDocked`		Dock the sql panel in the window or float it
Dock Shell	`panel.shell.toggleDocked`		Dock the shell panel in the window or float it

View › Show as

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Text ✓	`view.showAsText`		Editable source view
Markdown ✓	`view.markdownReading`		Rendered .md (toggle to/from source)
Hex ✓	`view.showAsBytes`		Hex / decimal / octal / binary editor
Table ✓	`view.showAsTable`		CSV / TSV / NDJSON as a table
Log ✓	`view.showAsLog`		Log view with follow + filter

Code

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Fold / Unfold	`fold.toggle`	`⌥⌘.`	Collapse or expand the region at the cursor

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Fold All	`fold.all`		Collapse every foldable region
Unfold All	`fold.unfoldAll`		Expand every collapsed region
Add Highlight Rule…	`decorations.addRule`		Color text matching a regex (saved, applies everywhere)
Clear Highlight Rules	`decorations.clearRules`		Remove all user highlight rules
Reopen with Encoding…	`encoding.reopen`		Re-decode the file as a chosen text encoding
Convert to UTF-8	`encoding.convertToUTF8`		Transcode a legacy/UTF-16 file to editable UTF-8
Format Document	`format.document`	⌥⇧F	Pretty-print JSON / XML / TOML
Minify Document	`format.minify`		Compact JSON / XML
Trim Trailing Whitespace	`transform.trimTrailing`		Remove spaces/tabs at the end of every line
Collapse Whitespace	`transform.collapseWhitespace`		Collapse runs of spaces/tabs to one space (confirms first)
Join Lines	`transform.joinLines`		Remove all line breaks (confirms first)
Sort Selected Lines	`lines.sort`		Sort the selected lines A→Z (case-insensitive)
Remove Duplicate Lines	`lines.dedup`		Drop repeated lines in the selection, keeping the first
Filter Through Command…	`transform.filterCommand`		Pipe the selection (or document) through a shell command, replacing it with the output
Validate Against JSON Schema…	`schema.validate`		Check the open JSON against a schema you choose

AI

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Edit Selection with AI…	`ai.editSelection`		Rewrite the selection, review as a diff
AI Key Status	`ai.keyStatus`		Selected provider and whether its key is set/verified
Choose AI Provider…	`ai.chooseProvider`		Anthropic · OpenAI-compatible · Ollama (local)
Set API Key…	`ai.setApiKey`		Store the selected provider's key in the Keychain

Help

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Check for Updates…	`help.checkForUpdates`		Compare this version against the latest release
Acknowledgements	`help.acknowledgements`		Licenses of the third-party components Runic includes
Privacy Policy	`help.privacy`		What Runic stores locally and what leaves your machine

Palette and shortcuts only

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Show Next Tab	`window.nextTab`	`^Tab`	
Show Previous Tab	`window.previousTab`	`^⇧Tab`	
Merge All Windows	`window.mergeAll`		
Find…	`find.show`	`⌘F`	
Find and Replace…	`replace.toggle`	`⌥⌘F`	
Find Next	`find.next`	`⌘G`	

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Find Previous	`find.previous`	⇧⌘G	
Close Find	`find.close`	Esc	
Add Next Occurrence	`selection.addNextOccurrence`	⌘D	Select the word, or add the next match as another cursor (multi-cursor)
Add Cursor Above	`selection.addCaretAbove`		Add a cursor on the line above (⌥⇧↑); ⌥-drag for a block selection
Add Cursor Below	`selection.addCaretBelow`		Add a cursor on the line below (⌥⇧↓); ⌥-drag for a block selection
Settings…	`app.preferences`		
About Runic	`app.about`		Show the About window
Quit Runic	`app.quit`		Quit (prompts to save unsaved changes)
Filter Table…	`table.filter`		Show only rows containing text (table view)
Query Table… ✓	`table.query`		Run SQL over the CSV/NDJSON rows (docks below)
Follow (Tail) ✓	`log.toggleFollow`		Stick to the bottom as the file grows (log view)
Filter Log…	`log.setFilter`		Show only matching lines (log view)
Clear Log Filter	`log.clearFilter`		Back to all lines (log view)
Bytes: Hexadecimal ✓	`view.radix.hexadecimal`		
Bytes: Decimal ✓	`view.radix.decimal`		
Bytes: Octal ✓	`view.radix.octal`		
Bytes: Binary ✓	`view.radix.binary`		

ITEM	COMMAND	SHORTCUT	DESCRIPTION
Byte Insert Mode ✓	`bytes.toggleInsertMode`		Insert vs overwrite (byte view)
Group: Byte (1) ✓	`view.byteUnit.1`		
Group: Word (2) ✓	`view.byteUnit.2`		
Group: DWord (4) ✓	`view.byteUnit.4`		
Group: QWord (8) ✓	`view.byteUnit.8`		
Little-Endian (LSB first) ✓	`view.byteEndian.little`		
Big-Endian (MSB first) ✓	`view.byteEndian.big`		

105 commands. A ✓ marks a toggle — the menu item shows a checkmark when it is on (33 of them).

Keyboard shortcut reference

SHORTCUT	ITEM	WHERE	COMMAND
`Esc`	Close Find	Palette and shortcuts only	`find.close`
`⌘D`	Add Next Occurrence	Palette and shortcuts only	`selection.addNextOccurrence`
`⌘F`	Find…	Palette and shortcuts only	`find.show`
`⌘G`	Find Next	Palette and shortcuts only	`find.next`
`⌘L`	Go to Line / Offset…	View	`view.goToLine`
`⌘N`	New	File	`file.new`
`⌘O`	Open…	File	`file.open`
`⌘P`	Quick Open…	View	`nav.quickOpen`
`⌘S`	Save	File	`file.save`
`⌘T`	New Tab	File	`window.newTab`
`⌘W`	Close	File	`window.close`
`⇧⌘A`	Assistant	View › Panels	`ai.toggleChat`
`⇧⌘G`	Find Previous	Palette and shortcuts only	`find.previous`
`⇧⌘O`	Go to Symbol…	View	`nav.gotoSymbol`
`⇧⌘P`	Command Palette…	View	`command.palette`
`⇧⌘S`	Save As…	File	`file.saveAs`
`⌥Z`	Wrap Lines	View	`view.toggleWrap`
`⌥⌘.`	Fold / Unfold	Code	`fold.toggle`
`⌥⌘F`	Find and Replace…	Palette and shortcuts only	`replace.toggle`
`⌥⌘M`	Minimap	View	`view.toggleMinimap`
`⌥⌘O`	Navigator	View › Panels	`view.outline`

SHORTCUT	ITEM	WHERE	COMMAND
`⌥⌘P`	Print…	File	`file.print`
`⌥⇧F`	Format Document	Code	`format.document`
`⌥⇧⌘O`	Open Folder…	File	`workspace.openFolder`
`^Tab`	Show Next Tab	Palette and shortcuts only	`window.nextTab`
`^⌘G`	Go to Byte Offset…	View	`nav.gotoByte`
`^⌘I`	Inspector	View › Panels	`view.inspector`
`^⌘L`	Lock Document	File	`file.toggleLock`
`^⌘T`	New Terminal	File	`terminal.new`
`^⇧Tab`	Show Previous Tab	Palette and shortcuts only	`window.previousTab`

30 of 105 commands have a default key binding. Every command can be bound to one — see Customising keyboard shortcuts.