

# IWE

## Installation

### How to Install

Installation instructions are below. Editor integration is covered in the quick start section.

### Using Homebrew (macOS/Linux)

The easiest way to install IWE on macOS or Linux is using Homebrew:

```
brew tap iwe-org/iwe
brew install iwe
```

This installs both the CLI (iwe) and the LSP server (iwes).

### From Crates.IO

- Rust and Cargo must be installed on your system. You can get them from [rustup.rs](https://rustup.rs).

IWE is available at [crates.io](https://crates.io). You can install IWE using cargo (and iwes for LSP server)

```
cargo install iwe
cargo install iwes
```

The binaries will be installed to `$HOME/.cargo/bin`. You may need to add it to your `$PATH`.

### From Source

Clone the repository, navigate into the project directory, and build the project:

```
git clone git@github.com:iwe-org/iwe.git
cd iwe
cargo build --release
```

This will create executables located in the `target/release` directory.

## Usage Guide

### How to Use with Your Text Editor

#### Purpose

This page intends to teach you how to trigger IWE's features from inside your text editor.

#### Background

IWE's features are implemented as Language Server Protocol (**LSP**) capabilities. This makes IWE *editor-agnostic*; it's intended to work the same across all text editors that support the LSP standard.










What this means for you is that to interact with IWE from inside your editor, you need to use *LSP requests*. These may be accessed differently across editors, and it's up to each editor to implement them properly. **If you've ever used something like "Find References" or "Go To Definition" before, then you're already familiar with LSP requests.**

#### Primary Features

IWE provides comprehensive features for markdown-based knowledge management:

#### Core LSP Features

- 🛠️ **Text Transformation**: Generate, rewrite, or modify text using configurable CLI commands
- 🔍 **Global Search**: Search through all notes using fuzzy matching on document paths and content
- 📍 **Link Navigation**: Follow links between documents with Go To Definition

-  **Hover Preview:** Preview linked notes without navigating away
-  **Extract/Inline Notes:** Split sections into separate files or merge them back
-  **Auto-Format:** Normalize document structure, headers, lists, and link titles
-  **Rename Refactoring:** Rename files while automatically updating all references
-  **Backlinks Discovery:** Find all documents that reference the current document
-  **Inlay Hints:** Display parent document references and link usage counts
-  **Auto-Complete:** Smart completion for links as you type
-  **Document Symbols:** Navigate document outline via table of contents
-  **Text Manipulation:** Transform lists to headers and vice-versa, change list types

## LSP Feature Reference

Here's a reference connecting each LSP request with IWE features:

IWE Feature	LSP Request	Description
Extract/Inline Notes	Code Action	Split sections into files or merge them back
Text Transformation	Code Action	Generate, rewrite, or modify text using CLI commands
Text Transformation	Code Action	Convert lists to headers, change list types
Link Navigation	Go To Definition	Follow markdown links to target documents
Hover Preview	Hover	Preview linked notes without opening the file
Backlinks	Go To References	Find all documents referencing current document
Document Outline	Document Symbols	View table of contents for navigation
Global Search	Workspace Symbols	Search through all notes with fuzzy matching
Auto-Format	Document Formatting	Normalize structure, headers, and links
File Renaming	Rename Symbol	Rename files and update all references
Link Completion	Completion	Auto-complete links as you type
Visual Hints	Inlay Hints	Show parent references and link counts

## Usage Example

**Editor Compatibility:** Most editors have keybindings for LSP requests. Common patterns include:

- VS Code: Ctrl+Shift+P (Command Palette) → search for LSP commands
- Neovim: <leader>ca (code actions), gd (go to definition), gr (references)
- Helix: space+a (code actions), gd (go to definition), gr (references)
- Zed: Cmd+. (code actions), F12 (go to definition), Shift+F12 (references)

Suppose that you have the following in a Markdown file:

**# My First Note**

There's some content here.

**## Another section**

With a list inside it:

- list item
- another item

### Extracting a Section

1. Move your cursor to the `## Another section` line
2. Invoke the **Code Action** command (varies by editor)
3. Select “Extract section” from the options
4. Your file will now look like this:

```
# My First Note
```

There's some content here.

```
[Another section](2sbdlvhe)
```

The 2sbdlvhe refers to the name of a new file IWE generated for you.

### Following the Link

1. Move your cursor anywhere on the `[Another section](2sbdlvhe)` link
2. Use **Go To Definition** command
3. Your editor will open the new file containing:

```
# Another section
```

With a list inside it:

- list item
- another item

### Finding Backlinks

1. In the extracted file, move your cursor to the `# Another section` line
2. Use the **Go To References** command
3. You’ll see a list of all files that link to this document
4. Select the original file to navigate back

## Advanced Features

### Text Transform Commands

IWE supports configurable text transformation commands that can:

- Rewrite and improve text
- Generate new content based on prompts
- Expand on ideas and concepts
- Add formatting and structure

Commands work by piping content through external CLI tools. Configure them in your `.iwe/config.toml`:

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120
```

```
[actions.rewrite]
```

```
type = "transform"
title = "Improve Text"
command = "claude"
input_template = "Improve this text: {{context}}"
```

You can use any CLI tool that reads from stdin and writes to stdout, including external tools like Claude CLI, custom scripts, or Unix commands.

## Configuration

IWE projects are configured through a `.iwe/config.toml` file in your project root. Below are all available configuration options.

### ##### Basic Configuration

```
[markdown]
refs_extension = ""
date_format = "%b %d, %Y"

[library]
path = ""
date_format = "%Y-%m-%d"
frontmatter_document_title = "title"

[completion]
link_format = "markdown"
```

### ##### Markdown Settings

- `refs_extension`: File extension for markdown references (default: empty, uses `.md`)
- `date_format`: Date format for markdown content display (default: `"%b %d, %Y"`, e.g., "Jan 15, 2024")

### ##### Library Settings

- `path`: Subdirectory for markdown files relative to project root (default: empty, uses root)
- `date_format`: Date format for file key generation (default: `"%Y-%m-%d"`, e.g., "2024-01-15")
- `frontmatter_document_title`: YAML frontmatter field to use as document title (default: none, uses first header)

### ##### Completion Settings

- `link_format`: Format for auto-completed links (default: "markdown")
  - "markdown": Creates `[title](key)` style links
  - "wiki": Creates `[[key]]` style WikiLinks

### ##### Date Format Patterns

Date formats use chrono format specifiers:

- `%Y`: 4-digit year (2024)
- `%y`: 2-digit year (24)
- `%m`: Month as number (01-12)
- `%b`: Abbreviated month name (Jan)
- `%B`: Full month name (January)
- `%d`: Day of month (01-31)
- `%A`: Full weekday name (Monday)
- `%a`: Abbreviated weekday name (Mon)

#### ##### Frontmatter Document Title

By default, IWE uses the first header in a document as its title for links, autocomplete suggestions, and search results. You can override this behavior by specifying a YAML frontmatter field to use instead:

```
[library]
frontmatter_document_title = "title"
```

With this configuration, a document like:

```
---
title: My Custom Title
---
```

#### # Header (ignored for title)

Document content...

Will use “My Custom Title” as the document title instead of “Header (ignored for title)”. This affects:

- Link text in auto-completed links: [My Custom Title](document-key)
- Link text normalization when references are updated
- Document titles in search results and workspace symbols

If the configured frontmatter field is missing or the document has no frontmatter, IWE falls back to using the first header as the title.

#### ##### Commands

Define CLI commands for text transformation actions. Commands receive input via stdin and output transformed content to stdout:

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120

[commands.uppercase]
run = "tr '[:lower:]' '[:upper:]'"
timeout_seconds = 5

[commands.custom_script]
run = "/path/to/my-script.sh"
timeout_seconds = 60
```

Each command requires:

- run: Command to execute (by default runs via `sh -c`)

Optional parameters:

- args: Array of arguments when using direct execution (only used when `shell = false`)
- cwd: Working directory for command execution
- env: Environment variables as key-value pairs (supports `$VAR` or `${VAR}` expansion from parent environment)
- shell: Execute via shell (`true`, default) or directly (`false`)
- timeout\_seconds: Maximum execution time in seconds (default: 120)

Commands are executed with the processed input template piped to stdin. The command’s stdout becomes the replacement content.

##### Example Commands

### Using Claude CLI:

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120
```

### Using a custom script:

```
[commands.rewriter]
run = "python ~/scripts/rewrite.py"
timeout_seconds = 30
```

### Simple text transformation:

```
[commands.uppercase]
run = "tr '[:lower:]' '[:upper:]'"
timeout_seconds = 5
```

### Direct execution with arguments (no shell):

```
[commands.claude_direct]
run = "claude"
args = ["-p", "--model", "sonnet"]
shell = false
timeout_seconds = 120
```

### With environment variables:

```
[commands.custom_api]
run = "my-api-tool"
env = { API_KEY = "$MY_API_KEY", DEBUG = "true" }
timeout_seconds = 60
```

### With custom working directory:

```
[commands.project_script]
run = "./scripts/process.sh"
cwd = "/path/to/project"
timeout_seconds = 30
```

##### Transform Actions

Transform actions modify text content in-place using configured commands:

```
[actions.rewrite]
type = "transform"
title = "Rewrite"
command = "claude"
input_template = """
Here's a text that I'm going to ask you to edit. The text is marked with
{{context_start}}{{context_end}} tag.
```

The part you'll need to update is marked with {{update\_start}}{{update\_end}}.

```
{{context_start}}
{{context}}
{{context_end}}
```

```
Rewrite the given text to improve clarity and readability.
"""
```

Transform action parameters:

- type: Must be "transform"
- title: Display name in editor
- command: Reference to command configuration
- input\_template: Template for preparing stdin input

##### Attach Actions

Link content under cursor to another file, creating daily notes or collections:

```
[actions.today]
type = "attach"
title = "Add to Today"
key_template = "{{today}}"
document_template = "# {{today}}\n\n{{content}}\n"

[actions.weekly_review]
type = "attach"
title = "Add to Weekly Review"
key_template = "weekly-{{today}}"
document_template = "# Weekly Review - {{today}}\n\n## Notes\n\n{{content}}\n\n## Action Items\n\n- [ ] \n"
```

Attach action parameters:

- type: Must be "attach"
- title: Display name in editor code actions
- key\_template: Template for target file key (supports {{today}} variable)
- document\_template: Template for new document content (supports {{today}} and {{content}} variables)

##### Template Variables

**Attach Actions** support:

- {{today}}: Current date formatted using library.date\_format (for keys) or markdown.date\_format (for content)
- {{content}}: The content being attached

**Transform Actions** support:

- {{context}}: Document context with the target block marked
- {{context\_start}}, {{context\_end}}: Context delimiters
- {{update\_start}}, {{update\_end}}: Update region delimiters

##### Examples

**Daily Note Creation**

```
[actions.daily]
type = "attach"
title = "Add to Daily Note"
key_template = "daily/{{today}}"
document_template = """"# Daily Note - {{today}}
```

```
## Today's Focus
```

```
{{content}}
```

```
## Tasks
- [ ]
```

```
## Notes
```

```
"""
```

## Project Collection

```
[actions.project_ideas]
type = "attach"
title = "Add to Project Ideas"
key_template = "projects/ideas"
document_template = "# Project Ideas\n\n{{content}}\n"
```

## Text Transformation with Claude CLI

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120
```

```
[actions.expand]
type = "transform"
title = "Expand"
command = "claude"
input_template = """
Here's a text that I'm going to ask you to edit. The text is marked with
{{context_start}}{{context_end}} tag.
```

The part you'll need to update is marked with {{update\_start}}{{update\_end}}.

```
{{context_start}}
{{context}}
{{context_end}}
```

Expand the text you need to update, generate a couple paragraphs.

```
"""
```

## Simple Text Transformation

```
[commands.uppercase]
run = "tr '[:lower:]' '[:upper:]'"
timeout_seconds = 5
```

```
[actions.uppercase]
type = "transform"
title = "UPPERCASE"
command = "uppercase"
input_template = "{{context}}"
```

## ##### Migration from Version 2

If you're upgrading from a configuration using the old [models] section, IWE will automatically migrate your configuration to version 3. The migration:

1. Renames [models] section to [commands] with empty run values
2. Renames model field to command in transform actions
3. Renames prompt\_template field to input\_template in transform actions
4. Removes the context field from transform actions



After migration, you'll need to manually update the run field in each command to specify the actual CLI command to execute.

#### Before (version 2):

```
version = 2

[models.default]
api_key_env = "OPENAI_API_KEY"
base_url = "https://api.openai.com"
name = "gpt-4o"

[actions.rewrite]
type = "transform"
title = "Rewrite"
model = "default"
prompt_template = "..."
context = "Document"
```

#### After (version 3):

```
version = 3

[commands.default]
run = "claude -p" # Update this to your preferred CLI command
timeout_seconds = 120

[actions.rewrite]
type = "transform"
title = "Rewrite"
command = "default"
input_template = "..."
```

### Text Transformations

Use **Code Actions** to transform document structure:

- Convert bullet lists to numbered lists
- Transform lists into header hierarchies
- Convert headers back to lists
- Change outline organization

### Auto-Formatting

The **Document Formatting** command will:

- Normalize header formatting and spacing
- Standardize list formatting
- Update link titles automatically
- Fix markdown syntax issues
- Ensure consistent document structure

### Global Search

Use **Workspace Symbols** to:

- Search across all documents
- Find content by fuzzy matching
- Navigate to specific sections
- Explore document relationships

Results show full paths like:

Journal, 2025 ⇒ Week 3 - Coffee week ⇒ Jan 26, 2025 - Cappuccino  
My Coffee Journey ⇒ Week 3 - Coffee week ⇒ Jan 26, 2025 - Cappuccino

## Inlining Extracted Sections

To reverse section extraction:

1. Move your cursor to a link like [Another section](2sbdlvhe)
2. Invoke **Code Action**
3. Select “Inline section”
4. The content returns to the original document:

```
# My First Note
```

There's some content here.

```
## Another section
```

With a list inside it:

- list item
- another item

**Note:** Inlining automatically deletes the separate file after merging content back.

## Working with New Files

When IWE creates new files (via extraction):

- Files are initially created in memory/buffer
- Save them using your editor's save command
- In some editors, use “Save All” to ensure all new files are written to disk
- Files use unique identifiers as filenames for reliable linking

## Best Practices

1. **Use Meaningful Headers:** Clear section titles improve navigation and search
2. **Link Liberally:** Create connections between related concepts
3. **Regular Formatting:** Use document formatting to maintain consistency
4. **Organize with Extraction:** Break large documents into focused, linked sections
5. **Leverage Search:** Use global search to discover connections and content
6. **Configure Commands:** Set up transform actions that match your writing workflow
7. **Use Inlay Hints:** Enable hints to understand document relationships at a glance

## CLI Commands

### IWE Init

Initializes the current directory as an IWE project.

### Usage

```
iwe init
```

### What It Creates

Running `iwe init` creates:

```
.iwe/  
└─ config.toml    # Project configuration
```

The `.iwe/` directory marks the project root and stores configuration files.

## Default Configuration

The generated `config.toml` contains:

```
version = 3

[markdown]
refs_extension = ""
date_format = "%b %d, %Y"

[library]
path = ""
date_format = "%Y-%m-%d"

[completion]

[commands]
[commands.default]
run = "claude -p"
timeout_seconds = 120

[actions]
[actions.extract]
type = "extract"
title = "Extract"
link_type = "markdown"
key_template = "{{id}}"

[actions.inline_section]
type = "inline"
title = "Inline section"
inline_type = "section"
keep_target = false

[actions.sort]
type = "sort"
title = "Sort A-Z"
reverse = false

[templates]
[templates.default]
key_template = "{{slug}}"
document_template = "# {{title}}\n\n{{content}}"
```

## Configuration Options

### Library Section

Option	Default	Description
path	""	Subdirectory containing markdown files (empty = project root)
date_format	%Y-%m-%d	Format for dates in document keys
default_template	null	Template name for iwe new command

## Markdown Section

Option	Default	Description
refs_extension	" "	File extension to append to references (e.g., .md)
date_format	%b %d, %Y	Format for dates displayed in documents

## Completion Section

Option	Default	Description
link_format	null	Link style for completions: markdown or wiki

## Example

```
cd ~/my-notes
iwe init
# Output: Creates .iwe/config.toml

# Verify initialization
ls -la .iwe/
# Output: config.toml
```

## Customization

After initialization, edit `.iwe/config.toml` to:

- Store markdown files in a subdirectory: set `library.path = "docs"`
- Use wiki-style links: set `completion.link_format = "wiki"`
- Define custom templates for document creation
- Configure AI-powered actions with custom commands

## Re-initialization

Running `iwe init` in an already-initialized project is safe - it will not overwrite an existing `config.toml`.

## IWE New

Creates a new document from a template.

## Usage

```
iwe new <TITLE> [OPTIONS]
```

## Arguments

- `<TITLE>`: Title for the new document (required)

## Options

- `-t, --template <NAME>`: Template name from config (default: "default")
- `-c, --content <CONTENT>`: Initial content for the document
- `-i, --if-exists <MODE>`: Behavior when file already exists (default: "suffix")
  - `suffix`: Append -1, -2, etc. to filename until unique
  - `override`: Overwrite existing file
  - `skip`: Do nothing, exit successfully without output
- `-e, --edit`: Open created file in `$EDITOR` after creation

## What it does

- Creates a new markdown file using the specified template
- Generates filename from the title (slugified)

- Supports content from command-line argument or stdin pipe
- Prints the absolute path of the created file to stdout
- Optionally opens the file in your configured editor

## Template Variables

Templates support the following variables:

- `{{title}}`: The provided title argument
- `{{slug}}`: Slugified title (kebab-case)
- `{{today}}`: Current date (uses `library.date_format` for key, `markdown.date_format` for content)
- `{{id}}`: Random 8-character alphanumeric ID
- `{{content}}`: Content from `-c` option or stdin

## Examples

```
# Create a new note with default template
iwe new "My New Note"
# Creates: my-new-note.md (or my-new-note-1.md if exists)

# Create with content
iwe new "Meeting Notes" --content "Discussed project timeline"

# Pipe content from clipboard (macOS)
pbpaste | iwe new "Clipboard Note"

# Create and open in editor
iwe new "Quick Idea" --edit

# Use a custom template
iwe new "Daily Journal" --template journal

# Overwrite existing file
iwe new "My Note" --if-exists override

# Skip if file exists (useful in scripts)
iwe new "My Note" --if-exists skip
```

## Configuration

Templates are defined in `.iwe/config.toml`:

```
[library]
default_template = "default" # Optional: set default template

[templates.default]
key_template = "{{slug}}"
document_template = "# {{title}}\n\n{{content}}"

[templates.journal]
key_template = "journal/{{today}}"
document_template = "# {{today}}\n\n{{content}}"
```

## IWE Retrieve

Retrieves document content with graph expansion and relationship context. Designed for AI agents to navigate and understand the knowledge graph.

## Usage

```
iwe retrieve [OPTIONS]
```

Without `-k`, reads the document key from stdin (for piping).

## Options

Flag	Description	Default
<code>-k, --key &lt;KEY&gt;</code>	Document key(s) to retrieve (can be specified multiple times)	stdin
<code>-e, --exclude &lt;KEY&gt;</code>	Exclude document key(s) from results (can be specified multiple times)	none
<code>-d, --depth &lt;N&gt;</code>	Follow children down N levels	1
<code>-c, --context &lt;N&gt;</code>	Include N levels of parent context	1
<code>-l, --links</code>	Include inline referenced documents	false
<code>-b, --backlinks</code>	Include incoming references in output	true
<code>-f, --format &lt;FMT&gt;</code>	Output format: markdown, keys, json	markdown
<code>--no-content</code>	Exclude content, show child documents instead (metadata only)	false
<code>--dry-run</code>	Show document count and total lines without content	false

## How It Works

The retrieve command collects documents in a specific order:

1. **Main document** - The document specified by `-k`
2. **Children** (`-d N`) - Documents included via Inclusion Links, expanded N levels deep
3. **Context** (`-c N`) - Parent documents of the main document, up N levels
4. **Sub-document parents** - Parents of direct sub-documents (only when both `-d > 0` and `-c > 0`)
5. **Links** (`-l`) - Documents referenced inline within the main document

Documents are deduplicated - each document appears only once in the output.

## Depth Expansion (-d)

Controls how deep to follow children (sub-documents):

- **-d 0**: Only the main document, no expansion
- **-d 1** (default): Main document + direct sub-documents
- **-d 2**: Main + sub-documents + their sub-documents
- **-d N**: Follow children up to N levels deep

## Context Expansion (-c)

Controls how many levels of parent documents to include:

- **-c 0**: No parent context documents included
- **-c 1** (default): Include direct parents of main document and direct parents of immediate sub-documents
- **-c N**: Include parent documents up to N levels up

## Sub-document Parent Context

When you retrieve a document with both depth (`-d`) and context (`-c`) enabled, the command also fetches parent documents of any sub-documents. This provides important context: the parent document often defines what *type* of thing the sub-document is.

## Example: A component used in multiple projects

Consider a button component embedded in two different projects. When you retrieve `mobile-app`:

```
iwe retrieve -k mobile-app -d 1 -c 1
```

The output for the button sub-document shows its other parents:

```
# Button
```

```
[Button](button) <- [Web Dashboard](web-dashboard)
```

A reusable button component.

The <- notation lists other documents where button is embedded. Since we're already viewing button from within mobile-app, only web-dashboard is shown — revealing that this component is shared across projects.

### Result includes:

1. mobile-app — the main document
2. button — sub-document (via -d 1)
3. web-dashboard — another parent of button (via -c 1)

The web-dashboard document is fetched because it's a parent of button. This context helps understand what the component is and where else it's used.

**Note:** Only parents of *direct* sub-documents are included.

### Links (-l)

When enabled, includes documents that are referenced inline (not as inclusion links):

```
# Main Document
```

See [Related Topic](related-topic) for more details.

With -l, related-topic would be included in the output.

### Document Relationships

#### Parent Documents vs Back Links

**Parent Documents** - Inclusion links (document embedded in another):

```
# Parent Document
```

```
## Overview
```

```
[child](child) <- Inclusion link creates parent-child relationship
```

The child document shows: **Parent documents:** [Parent Document](parent) > Overview

**Back Links** - Inline references (links within text):

```
# Some Document
```

See [target](target) for more details. <- Inline reference creates backlink

The target document shows: **Back links:** [Some Document](some-document)

### Output Format

#### Markdown Format (default)

Documents are rendered with YAML frontmatter containing metadata, followed by the document content:

```

---
document:
  key: my-document
  title: My Document
  parents:
    - key: index
      title: Index
  back-links:
    - key: related-doc
      title: Related Doc
---

```

## # My Document

Original document content preserved exactly as written.

Each document in the result set includes:

- YAML frontmatter with key, title, parents, and back-links
- Document content with original headers preserved
- Two empty lines after each document for easy parsing

Multiple documents are separated by their frontmatter delimiters (---), no horizontal rules.

## Keys Format (-f keys)

```

my-document
child-document
parent-document

```

One key per line, suitable for piping to other commands or building exclude lists.

## JSON Format (-f json)

```

{
  "documents": [
    {
      "key": "my-document",
      "title": "My Document",
      "content": "# My Document\n\nContent here...",
      "parent_documents": [
        {
          "key": "index",
          "title": "Index",
          "section_path": ["Topics", "My Topic"]
        }
      ],
      "backlinks": [
        {
          "key": "related-doc",
          "title": "Related Doc",
          "section_path": []
        }
      ]
    }
  ]
}

```



## Dry Run (--dry-run)

Shows statistics without outputting content:

```
$ iwe retrieve -k my-document --dry-run
documents: 5
lines: 234
```

Useful for checking how much content would be retrieved before fetching it.

## Examples

```
# Retrieve single document with defaults (depth=1, context=1)
iwe retrieve -k my-document

# Retrieve multiple documents at once
iwe retrieve -k doc1 -k doc2 -k doc3

# Retrieve only the main document, no expansion
iwe retrieve -k my-document -d 0 -c 0

# Retrieve with deep expansion (3 levels of sub-documents)
iwe retrieve -k my-document -d 3

# Include more parent context
iwe retrieve -k my-document -c 2

# Include inline referenced documents
iwe retrieve -k my-document -l

# Exclude documents which you don't need
iwe retrieve -k my-document -e already-loaded -e another-loaded

# Get metadata only, no content
iwe retrieve -k my-document --no-content

# Check size before retrieving
iwe retrieve -k my-document -d 2 -c 1 --dry-run

# Get JSON output for programmatic processing
iwe retrieve -k my-document -f json

# Pipe keys from stdin (one per line)
echo -e "doc1\ndoc2\ndoc3" | iwe retrieve

# Without backlinks (cleaner output)
iwe retrieve -k my-document -b false
```

## Use Cases

### AI Agent Context Building

Build rich context for AI agents navigating the knowledge base:

```
# Get document with immediate context
iwe retrieve -k authentication

# Check context size first
iwe retrieve -k authentication -d 2 --dry-run
```

```
# Get broader context if needed
iwe retrieve -k authentication -d 2 -c 2
```

### Understanding Document Relationships

```
# See where a document is used (parent documents in output)
iwe retrieve -k my-topic -d 0 -c 0
```

```
# See the full context including inline references
iwe retrieve -k my-topic -l
```

### Exploring Knowledge Base Structure

```
# Start from an entry point and expand
iwe retrieve -k index -d 2

# Get JSON for analysis
iwe retrieve -k project-overview -d 1 -f json
```

### Chaining Retrievals

Use the keys format to chain retrieval operations and exclude already-fetched documents:

```
# Get keys from first retrieval
KEYS_A=$(iwe retrieve -k topic-a -f keys)

# Retrieve for topic-b, excluding keys from topic-a
iwe retrieve -k topic-b -e $KEYS_A

# Or using command substitution with xargs
iwe retrieve -k topic-b $(iwe retrieve -k topic-a -f keys | xargs -I {} echo "-e {}")
```

### Technical Notes

- Documents use YAML frontmatter for metadata, content follows with original formatting
- Empty parents or back-links fields are omitted from frontmatter
- Original document headers are preserved (no level shifting)
- Two empty lines separate documents for easier parsing
- Duplicate documents are automatically filtered out
- Sub-document parent context only includes parents of direct (first-level) sub-documents, not nested ones

### IWE Find

Search and discover documents in your knowledge base with fuzzy matching and relationship filtering.

### Usage

```
iwe find [QUERY] [OPTIONS]
```

### Options

Flag	Description	Default
[QUERY]	Fuzzy search on document title and key	none (lists all)
--roots	Only show root documents (no parents)	false
--refs-to <KEY>	Documents that reference this key	none
--refs-from <KEY>	Documents referenced by this key	none
-l, --limit <N>	Maximum number of results	50

Flag	Description	Default
-f, --format <FMT>	Output format: markdown, keys, json	markdown

## How It Works

The find command searches and filters documents in your knowledge base:

1. **Fuzzy matching** - Uses the same fuzzy search algorithm as the LSP server (SkimMatcherV2)
2. **Ranking** - Without a query, documents are sorted by popularity (incoming references count)
3. **Filtering** - Apply filters for root documents or reference relationships
4. **Parent context** - Results include parent document information

## Fuzzy Search

The fuzzy matcher searches across both the document key and title:

```
# Finds "authentication.md" with title "User Authentication"
iwe find auth
```

```
# Finds documents with "api" in key or title
iwe find api
```

## Root Documents

Root documents are entry points - documents with no parents:

```
# List only root documents (no parents)
iwe find --roots
```

## Reference Filters

Find documents based on their relationships:

```
# Documents that reference "authentication"
iwe find --refs-to authentication
```

```
# Documents referenced by "index"
iwe find --refs-from index
```

## Output Formats

### Markdown Format (default)

```
## Documents
```

Found 3 results:

- [User Authentication](authentication) (root)
- [Login Flow](login-flow) <- [User Authentication](authentication)
- [Session Management](session-management) <- [User Authentication](authentication)

Each result shows:

- Document title and key as a markdown link
- (root) indicator if no parents
- Parent documents shown with <- arrow

### Keys Format (-f keys)

```
authentication
login-flow
session-management
```

One key per line, suitable for piping to other commands.

### JSON Format (-f json)

```
{
  "query": "auth",
  "total": 3,
  "results": [
    {
      "key": "authentication",
      "title": "User Authentication",
      "is_root": true,
      "incoming_refs": 5,
      "outgoing_refs": 2,
      "parent_documents": []
    },
    {
      "key": "login-flow",
      "title": "Login Flow",
      "is_root": false,
      "incoming_refs": 2,
      "outgoing_refs": 0,
      "parent_documents": [
        {
          "key": "authentication",
          "title": "User Authentication",
          "section_path": ["Implementation"]
        }
      ]
    }
  ]
}
```

Fields:

- query - The search query (null if no query provided)
- total - Total matching documents (before limit applied)
- results - Array of matching documents
  - is\_root - True if no parents
  - incoming\_refs - Count of parents + inline references to this document
  - outgoing\_refs - Count of children in this document
  - parent\_documents - Documents that include this one

### Examples

```
# List all documents (sorted by popularity)
iwe find
```

```
# Fuzzy search for documents
iwe find authentication
iwe find "api endpoint"
```

```
# List only root documents (entry points)
iwe find --roots
```

```
# Find what references a specific document
iwe find --refs-to my-document
```

```
# Find what a document references
iwe find --refs-from index

# Combine search with filters
iwe find api --roots

# Limit results
iwe find --limit 10

# Get JSON for programmatic use
iwe find -f json

# Get keys for piping
iwe find --roots -f keys

# Pipe keys to retrieve command
iwe find --roots -f keys | head -5 | xargs -I {} iwe retrieve -k {}
```

## Use Cases

### Discover Entry Points

Find root documents that serve as entry points to different topics:

```
iwe find --roots
```

### Explore Document Relationships

See what documents reference or are referenced by a specific document:

```
# What uses this document?
iwe find --refs-to authentication

# What does this document use?
iwe find --refs-from index
```

### Quick Document Lookup

Fuzzy search when you remember part of a document name:

```
iwe find deploy    # Finds "deployment", "deploy-script", etc.
iwe find config    # Finds "configuration", "config-options", etc.
```

### Pipeline Integration

Use keys format for scripting:

```
# Retrieve content for top 5 root documents
iwe find --roots -l 5 -f keys | while read key; do
    iwe retrieve -k "$key" -d 0
done

# Export all root documents to separate files
iwe find --roots -f keys | xargs -I {} sh -c 'iwe retrieve -k {} > {}.out.md'
```

### Analyze Knowledge Base Structure

Use JSON output for analysis:

```
# Find most referenced documents
iwe find -f json | jq '.results | sort_by(-.incoming_refs) | .[0:5]'
```

```
# Find orphan documents (roots with no outgoing refs)
iwe find --roots -f json | jq '.results | map(select(.outgoing_refs == 0))'
```

## Technical Notes

- Without a query, documents are sorted by incoming reference count (popularity)
- With a query, results are sorted by fuzzy match score
- The limit is applied after sorting, so you get the top N results
- Parent documents show section path breadcrumbs when applicable
- Both Inclusion Links and inline references count toward incoming\_refs

## IWE Normalize

Performs comprehensive document normalization across all markdown files in your knowledge base.

## Usage

```
iwe normalize
```

## Operations Performed

Operation	Description
Link title sync	Updates link text to match target document headers
Header leveling	Adjusts header levels for consistent hierarchy
List renumbering	Fixes ordered list numbering (1, 2, 3...)
Whitespace normalization	Standardizes newlines and indentation
List formatting	Ensures consistent list item formatting
Structure cleanup	Removes redundant empty lines

## Before/After Examples

### Link Title Sync

Before:

See the [old title](project-docs) for details.

After (if project-docs.md has header # Project Documentation):

See the [Project Documentation](project-docs) for details.

### List Renumbering

Before:

1. First item
1. Second item
5. Third item

After:

1. First item
2. Second item
3. Third item

### Whitespace Normalization

Before:

```
# Title
```

Some paragraph.

Another paragraph.

After:

**# Title**

Some paragraph.

Another paragraph.

## Examples

```
# Basic normalization
iwe normalize
```

```
# With INFO level logging
iwe -v 1 normalize
```

```
# With DEBUG level logging
iwe -v 2 normalize
```

## Safety Warning

**Important:** The normalize command modifies files in place. Always ensure you have a backup or version control before running:

```
# Recommended: commit changes before normalizing
git add -A && git commit -m "Before normalization"
```

```
# Run normalization
iwe normalize
```

```
# Review changes
git diff
```

## Configuration

Normalization behavior is controlled by `.iwe/config.toml`:

```
[markdown]
refs_extension = "" # Extension for reference links
```

## Idempotency

Running normalize multiple times produces the same result - once files are normalized, subsequent runs make no changes.

## IWE Tree

Display document hierarchy as a tree structure.

## Usage

```
iwe tree [OPTIONS]
```

## Options

Option	Default	Description
-f, --format <FORMAT>	markdown	Output format: markdown, keys, json
-k, --key <KEY>	-	Start tree from specific document(s), can be repeated
-d, --depth <DEPTH>	4	Maximum depth to traverse
-v, --verbose <LEVEL>	0	Verbosity level (1=info, 2=debug)

## Output Formats

### Markdown (default)

Nested list with links:

- [Main Document](main)
  - [Child Document](child)
    - [Nested Item](nested)
- [Another Root](another)

### Keys

Document keys only with tab indentation:

```
main
  child
    nested
another
```

### JSON

Nested JSON array structure:

```
[
  {
    "key": "main",
    "title": "Main Document",
    "children": [
      {
        "key": "child",
        "title": "Child Document"
      }
    ]
  }
]
```

## Starting from Specific Documents

Use -k to start the tree from specific document(s):

```
iwe tree -k my-doc
iwe tree -k doc-a -k doc-b
```

This is essential for documents involved in circular references that have no natural root.

## Handling Circular References

When documents form circular references ( $A \rightarrow B \rightarrow C \rightarrow A$ ), they have no natural root and won't appear in the default tree output. Use -k to start from any document in the cycle:

```
iwe tree -k doc-a
```

Output shows the cycle:



- [Doc A](doc-a)
  - [Doc B](doc-b)
    - [Doc C](doc-c)
      - [Doc A](doc-a)

## Examples

```
iwe tree
iwe tree -f keys
iwe tree -f json
iwe tree -k my-doc
iwe tree -k doc-a -k doc-b
iwe tree --depth 2
iwe tree | grep -i api
iwe tree -f keys | grep cli
```

## Depth Impact

Depth	Shows
1	Root documents only
2	Roots and their direct children
3	Up to grandchildren
4+	Deeper nested relationships

## AI Agent Tips

- Use `tree` to understand the overall structure of a knowledge base
- Use `-f keys` for programmatic processing
- Use `-f json` for structured data consumption
- Pipe through `grep` to filter results
- Use `-k` to explore documents involved in circular references
- Use `--depth 2` to quickly identify major topic areas

## IWE Squash

Creates consolidated documents by combining linked content into a single markdown file.

## Usage

```
iwe squash <KEY> [OPTIONS]
```

## Arguments

Argument	Description
<KEY>	Document key to squash

## Options

Option	Default	Description
-d, --depth <DEPTH>	2	How deep to traverse links
-v, --verbose <LEVEL>	0	Verbosity level

## What It Does

1. Starts from the specified document
2. Traverses Inclusion Links up to specified depth
3. Combines content into a single markdown document

4. Converts linked sections to inline sections
5. Adjusts header levels to maintain hierarchy

## Output Format

Given this document structure:

project-overview.md:

### # Project Overview

Introduction to the project.

- [Goals](goals)
- [Architecture](architecture)

goals.md:

### # Goals

Our main objectives are:

- Improve performance
- Add new features

Running iwe squash project-overview outputs:

### # Project Overview

Introduction to the project.

#### ## Goals

Our main objectives are:

- Improve performance
- Add new features

#### ## Architecture

...

Note that linked document headers become sub-headers (# → ##) to preserve hierarchy.

## Examples

```
# Squash starting from document "project-overview"
iwe squash project-overview
```

```
# Squash with greater depth
iwe squash main-topic --depth 4
```

```
# Save output to file
iwe squash research-notes --depth 3 > output.md
```

```
# With debug output
iwe squash research-notes --depth 3 -v 2
```

## Header Level Adjustment

When documents are inlined, their header levels are adjusted:

Original Level	Becomes	Reason
# (h1)	## (h2)	Linked document becomes section
## (h2)	### (h3)	Preserves relative hierarchy
### (h3)	#### (h4)	And so on...

This ensures the squashed document maintains a logical structure with the root document as the top-level heading.

### Use Cases

- **Export to PDF:** Create a single document for typesetting with tools like Typst
- **Sharing:** Generate a standalone file from interconnected notes
- **Backup:** Consolidate project knowledge into a portable format
- **Context building:** Create comprehensive documents for review

### AI Agent Tips

- Use squash to build comprehensive context from related documents
- Combine with retrieve for maximum context: squash provides structure, retrieve provides backlinks
- Adjust depth based on scope: shallow (2) for focused topics, deep (4+) for comprehensive overviews
- Squashed output is ideal for sending to language models as context

Example PDF generated using squash command and Typst.

### IWE Stats

Generates comprehensive statistics about your knowledge graph.

### Usage

```
iwe stats [OPTIONS]
```

### Options

- `-f, --format <FORMAT>`: Output format (default: markdown)
  - markdown: Human-readable formatted statistics
  - csv: Machine-readable CSV format with per-document statistics

### What it shows

The stats command provides detailed analytics across multiple dimensions:

### Overview

- Total documents in your knowledge base
- Total nodes (all content elements)
- Total paths through the graph

### Document Statistics

- Total sections/headers across all documents
- Average sections per document
- Top 10 documents by section count
- Paragraph counts per document

### Reference Statistics

- Inclusion links (embedded documents)
- Inline references (wiki-links)

- Total references count
- Orphaned documents (no incoming references)
- Leaf documents (no outgoing references)
- Top 10 most referenced documents

### Lines Statistics

- Total lines across all documents
- Average lines per document
- Top 10 largest documents by line count

### Words Statistics

- Total words across all documents
- Average words per document
- Top 10 largest documents by word count

### Structure Statistics

- Root-level sections
- Maximum and average path depth
- Counts of bullet lists, ordered lists, code blocks, tables, and quotes

### Network Analysis

- Average references per document
- Top 10 most connected documents (by total incoming + outgoing references)

### Examples

```
# Generate human-readable statistics (default)
iwe stats

# Export per-document statistics as CSV
iwe stats --format csv > stats.csv

# Analyze CSV data with standard tools
iwe stats -f csv | cut -d, -f1,2,3 | column -t -s,
```

### Sample Markdown Output

#### # Graph Statistics

#### ## Overview

```
- **Total documents:** 39
- **Total nodes:** 1552
- **Total paths:** 302
```

#### ## Document Statistics

```
- **Total sections:** 844
- **Average sections/doc:** 21.64
```

#### ### Top Documents by Sections

```
1. **VS Code** (102 sections)
2. **Neovim** (89 sections)
3. **Extract Actions** (79 sections)
...
```

## ## Lines Statistics

- **Total lines:** 3404
- **Average lines/doc:** 87.28

## ### Top Documents by Lines

1. **Neovim** (429 lines)
2. **Extract Actions** (342 lines)
- ...

## ## Words Statistics

- **Total words:** 14204
- **Average words/doc:** 364.21

## ### Top Documents by Words

1. **Neovim** (1337 words)
2. **Extract Actions** (1200 words)
- ...

## CSV Format Details

The CSV format provides per-document statistics with the following columns:

- key: Document identifier/filename
- title: Document title (first heading)
- sections: Number of heading sections
- paragraphs: Number of paragraph blocks
- lines: Total line count
- words: Total word count
- parents: Documents that include this one
- incoming\_inline\_refs: Inline wiki-links pointing to this document
- total\_incoming\_refs: Total incoming references
- children: Documents included by this one
- outgoing\_inline\_refs: Inline wiki-links from this document
- total\_connections: Total references (incoming + outgoing)
- bullet\_lists: Number of unordered lists
- ordered\_lists: Number of numbered lists
- code\_blocks: Number of code/raw blocks
- tables: Number of tables
- quotes: Number of quote blocks

## Using CSV Output

The CSV format enables programmatic analysis and integration with data tools:

```
# Import into spreadsheet applications
iwe stats -f csv > knowledge-base-stats.csv
```

```
# Find most connected documents
iwe stats -f csv | tail -n +2 | sort -t, -k12 -nr | head -5
```

```
# Calculate total word count
```

```
iwe stats -f csv | tail -n +2 | cut -d, -f6 | paste -sd+ | bc
```

```
# Filter documents with many references
```

```
iwe stats -f csv | awk -F, '$9 > 5 {print $1, $2, $9}' OFS=,
```

```
# Generate reports with Python/pandas
```

```
import pandas as pd
```

```
df = pd.read_csv('stats.csv')
```

```
print(df.describe())
```

```
print(df.nlargest(10, 'total_connections'))
```

## IWE Export

Exports graph structure in various formats for visualization and analysis.

## Usage

```
iwe export [OPTIONS] <FORMAT>
```

## Available Formats

Format	Description
dot	Graphviz DOT format for graph visualization

## Options

Option	Default	Description
-k, --key <KEY>	all roots	Filter to specific document and its connections
-d, --depth <DEPTH>	0	Maximum depth to include (0 = unlimited)
--include-headers	false	Include section headers and create detailed subgraphs
-v, --verbose <LEVEL>	0	Verbosity level

## DOT Output Format

The DOT format produces Graphviz-compatible output:

```
digraph G {
    rankdir="LR"
    fontname="Verdana"
    fontsize="13"
    nodesep="0.7"
    splines="polyline"
    pad="0.5,0.2"
    ranksep="1.2"
    overlap="false"
    0 [label="Project Overview" fillcolor="#e8f0e8" fontsize="24" fontname="Verdana"
color="#b3b3b3" penwidth="1.5" shape="note" style="filled"]
    1 [label="Goals" fillcolor="#e8f0e8" fontsize="16" fontname="Verdana"
color="#b3b3b3" penwidth="1.5" shape="note" style="filled"]
    2 [label="Architecture" fillcolor="#e8f0e8" fontsize="16" fontname="Verdana"
color="#b3b3b3" penwidth="1.5" shape="note" style="filled"]
    0 -> 1 [color="#38546c66" arrowhead="normal" penwidth="1.2"]
    0 -> 2 [color="#38546c66" arrowhead="normal" penwidth="1.2"]
}
```

Nodes represent documents, edges represent links between them.

## Examples

```
# Export entire graph
iwe export dot

# Export specific document and connections
iwe export dot --key "project-main"

# Include section headers for detailed view
iwe export dot --include-headers

# Export with depth limit
iwe export dot --key "research" --depth 3

# Export with headers and depth limit
iwe export dot --key "research" --depth 3 --include-headers
```

## Generating Images

```
# Generate PNG visualization
iwe export dot > graph.dot
dot -Tpng graph.dot -o graph.png

# Generate SVG for web use
iwe export dot --include-headers > detailed.dot
dot -Tsvg detailed.dot -o detailed.svg

# Direct to PNG (one-liner)
iwe export dot | dot -Tpng -o graph.png

# Interactive visualization in browser
iwe export dot | dot -Tsvg > graph.svg && open graph.svg
```

## Depth Behavior

Depth	Behavior
0	Unlimited - include all reachable documents
1	Only the specified document
2	Document and its direct links
3+	Document and N-1 levels of connections

## With vs Without Headers

Mode	Use Case
Without --include-headers	High-level document relationships
With --include-headers	Detailed view showing internal sections

## AI Agent Tips

- Use `export dot` to analyze document relationship topology
- Generate visualizations to identify disconnected clusters
- Use `--depth` to focus on specific neighborhoods in large graphs
- Combine with `--key` to visualize a single topic and its context
- The graph structure reveals how knowledge is organized and connected

## CLI Workflow Examples

Practical examples of using IWE CLI commands for common tasks.

### Daily Maintenance

```
iwe normalize
```

```
iwe tree --depth 5
```

### Content Analysis

```
iwe tree
```

```
iwe tree -f keys | grep api
```

```
iwe export dot --key "machine-learning" --include-headers > ml.dot  
dot -Tpng ml.dot -o ml-graph.png
```

### Document Consolidation

```
# Create comprehensive document from research notes  
iwe squash research-index --depth 4 > consolidated-research.md  
  
# Generate presentation material  
iwe squash project-summary --depth 2 > project-overview.md
```

### Large Library Management

```
iwe normalize -v 2  
  
iwe tree --depth 8 -v 2  
  
iwe export dot --include-headers --depth 5 > full-graph.dot
```

## CLI Troubleshooting

Best practices and solutions to common issues when using the IWE CLI.

### Best practices

1. **Start small:** Test commands on a few files before processing large libraries
2. **Backup first:** Always backup before running normalize or other bulk operations
3. **Use debug mode:** Add -v 2 to see detailed debug information
4. **Iterate gradually:** Use increasing depth values to explore graph complexity
5. **Visualize regularly:** Export graphs to understand document relationships
6. **Monitor root documents:** Use tree to track entry points as your library grows

### Common issues

Issue	Solution
No changes after normalize	Check that files are properly formatted markdown
Export produces no output	Verify documents contain links and references
Squash fails	Ensure the specified key exists and is accessible
Command not found	Ensure IWE is installed and available in your PATH
Permission denied	Check file permissions in your project directory

### Debugging

When encountering issues, use verbose mode to get more information:



```
# INFO level logging
iwe -v 1 <command>

# DEBUG level logging
iwe -v 2 <command>
```

## Edge Cases

### Empty Knowledge Base

When no markdown files exist:

Command	Behavior
tree	No output
find	No matches found
stats	Shows zero counts
export dot	Produces empty graph

### Missing Referenced Documents

When a document links to a non-existent file:

- **Normalize:** Updates link title to empty string
- **Retrieve:** Skips missing references in expansion
- **Squash:** Skips missing linked documents
- **Export:** Excludes edges to missing documents

### Circular References

IWE handles circular references gracefully:

- **Retrieve:** Expands each document once, avoiding infinite loops
- **Squash:** Includes each document once at first encounter
- **Tree:** Use `-k` to start from any document in a cycle
- **Export:** Renders cycles as valid graph edges

### Large Knowledge Bases

For repositories with thousands of files:

- Use `--depth` limits to constrain exploration
- Use `find` with filters for targeted searches
- Use `tree -k` to explore specific subtrees
- Consider exporting subgraphs with `--key` filter

### Getting help

For any command, use the `--help` flag to see available options:

```
iwe --help
iwe <command> --help
```

### Configuration

IWE projects are configured through a `.iwe/config.toml` file in your project root. Below are all available configuration options.

#### Basic Configuration

```
[markdown]
refs_extension = ""
```

```
date_format = "%b %d, %Y"

[library]
path = ""
date_format = "%Y-%m-%d"
frontmatter_document_title = "title"

[completion]
link_format = "markdown"
```

## Markdown Settings

- `refs_extension`: File extension for markdown references (default: empty, uses `.md`)
- `date_format`: Date format for markdown content display (default: `"%b %d, %Y"`, e.g., “Jan 15, 2024”)

## Library Settings

- `path`: Subdirectory for markdown files relative to project root (default: empty, uses root)
- `date_format`: Date format for file key generation (default: `"%Y-%m-%d"`, e.g., “2024-01-15”)
- `frontmatter_document_title`: YAML frontmatter field to use as document title (default: none, uses first header)

## Completion Settings

- `link_format`: Format for auto-completed links (default: `"markdown"`)
  - `"markdown"`: Creates `[title](key)` style links
  - `"wiki"`: Creates `[[key]]` style WikiLinks

## Date Format Patterns

Date formats use chrono format specifiers:

- `%Y`: 4-digit year (2024)
- `%y`: 2-digit year (24)
- `%m`: Month as number (01-12)
- `%b`: Abbreviated month name (Jan)
- `%B`: Full month name (January)
- `%d`: Day of month (01-31)
- `%A`: Full weekday name (Monday)
- `%a`: Abbreviated weekday name (Mon)

## Frontmatter Document Title

By default, IWE uses the first header in a document as its title for links, autocomplete suggestions, and search results. You can override this behavior by specifying a YAML frontmatter field to use instead:

```
[library]
frontmatter_document_title = "title"
```

With this configuration, a document like:

```
---
title: My Custom Title
---
```

**# Header (ignored for title)**

Document content...

Will use “My Custom Title” as the document title instead of “Header (ignored for title)”. This affects:

- Link text in auto-completed links: [My Custom Title](document-key)
- Link text normalization when references are updated
- Document titles in search results and workspace symbols

If the configured frontmatter field is missing or the document has no frontmatter, IWE falls back to using the first header as the title.

## Commands

Define CLI commands for text transformation actions. Commands receive input via stdin and output transformed content to stdout:

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120

[commands.uppercase]
run = "tr '[:lower:]' '[:upper:]'"
timeout_seconds = 5

[commands.custom_script]
run = "/path/to/my-script.sh"
timeout_seconds = 60
```

Each command requires:

- run: Command to execute (by default runs via `sh -c`)

Optional parameters:

- args: Array of arguments when using direct execution (only used when `shell = false`)
- cwd: Working directory for command execution
- env: Environment variables as key-value pairs (supports `$VAR` or `${VAR}` expansion from parent environment)
- shell: Execute via shell (`true`, default) or directly (`false`)
- timeout\_seconds: Maximum execution time in seconds (default: 120)

Commands are executed with the processed input template piped to stdin. The command’s stdout becomes the replacement content.

## Example Commands

### Using Claude CLI:

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120
```

### Using a custom script:

```
[commands.rewriter]
run = "python ~/scripts/rewrite.py"
timeout_seconds = 30
```

### Simple text transformation:

```
[commands.uppercase]
run = "tr '[:lower:]' '[:upper:]'"
timeout_seconds = 5
```

### Direct execution with arguments (no shell):

```
[commands.claude_direct]
run = "claude"
args = ["-p", "--model", "sonnet"]
shell = false
timeout_seconds = 120
```

### With environment variables:

```
[commands.custom_api]
run = "my-api-tool"
env = { API_KEY = "$MY_API_KEY", DEBUG = "true" }
timeout_seconds = 60
```

### With custom working directory:

```
[commands.project_script]
run = "./scripts/process.sh"
cwd = "/path/to/project"
timeout_seconds = 30
```

## Transform Actions

Transform actions modify text content in-place using configured commands:

```
[actions.rewrite]
type = "transform"
title = "Rewrite"
command = "claude"
input_template = ""
Here's a text that I'm going to ask you to edit. The text is marked with
{{context_start}}{{context_end}} tag.
```

The part you'll need to update is marked with {{update\_start}}{{update\_end}}.

```
{{context_start}}
{{context}}
{{context_end}}
```

```
Rewrite the given text to improve clarity and readability.
"""
```

Transform action parameters:

- type: Must be "transform"
- title: Display name in editor
- command: Reference to command configuration
- input\_template: Template for preparing stdin input

## Attach Actions

Link content under cursor to another file, creating daily notes or collections:

```
[actions.today]
type = "attach"
title = "Add to Today"
key_template = "{{today}}"
document_template = "# {{today}}\n\n{{content}}\n"
```

```
[actions.weekly_review]
```

```

type = "attach"
title = "Add to Weekly Review"
key_template = "weekly-{{today}}"
document_template = "# Weekly Review - {{today}}\n\n## Notes\n\n{{content}}\n\n##\nAction Items\n\n- [ ] \n"

```

Attach action parameters:

- type: Must be "attach"
- title: Display name in editor code actions
- key\_template: Template for target file key (supports {{today}} variable)
- document\_template: Template for new document content (supports {{today}} and {{content}} variables)

## Template Variables

**Attach Actions** support:

- {{today}}: Current date formatted using `library.date_format` (for keys) or `markdown.date_format` (for content)
- {{content}}: The content being attached

**Transform Actions** support:

- {{context}}: Document context with the target block marked
- {{context\_start}}, {{context\_end}}: Context delimiters
- {{update\_start}}, {{update\_end}}: Update region delimiters

## Examples

### Daily Note Creation

```

[actions.daily]
type = "attach"
title = "Add to Daily Note"
key_template = "daily/{{today}}"
document_template = """"# Daily Note - {{today}}

```

```

## Today's Focus

```

```

{{content}}

```

```

## Tasks

```

```

- [ ]

```

```

## Notes

```

```

""""

```

### Project Collection

```

[actions.project_ideas]
type = "attach"
title = "Add to Project Ideas"
key_template = "projects/ideas"
document_template = "# Project Ideas\n\n{{content}}\n"

```

## Text Transformation with Claude CLI

```

[commands.claude]
run = "claude -p"

```

```
timeout_seconds = 120
```

```
[actions.expand]
```

```
type = "transform"
```

```
title = "Expand"
```

```
command = "claude"
```

```
input_template = ""
```

```
Here's a text that I'm going to ask you to edit. The text is marked with  
{{context_start}}{{context_end}} tag.
```

```
The part you'll need to update is marked with {{update_start}}{{update_end}}.
```

```
{{context_start}}
```

```
{{context}}
```

```
{{context_end}}
```

```
Expand the text you need to update, generate a couple paragraphs.
```

```
""
```

## Simple Text Transformation

```
[commands.uppercase]
```

```
run = "tr '[:lower:]' '[:upper:]'"
```

```
timeout_seconds = 5
```

```
[actions.uppercase]
```

```
type = "transform"
```

```
title = "UPPERCASE"
```

```
command = "uppercase"
```

```
input_template = "{{context}}"
```

## Migration from Version 2

If you're upgrading from a configuration using the old [models] section, IWE will automatically migrate your configuration to version 3. The migration:

1. Renames [models] section to [commands] with empty run values
2. Renames model field to command in transform actions
3. Renames prompt\_template field to input\_template in transform actions
4. Removes the context field from transform actions

After migration, you'll need to manually update the run field in each command to specify the actual CLI command to execute.

### Before (version 2):

```
version = 2
```

```
[models.default]
```

```
api_key_env = "OPENAI_API_KEY"
```

```
base_url = "https://api.openai.com"
```

```
name = "gpt-4o"
```

```
[actions.rewrite]
```

```
type = "transform"
```

```
title = "Rewrite"
```

```
model = "default"
```

```
prompt_template = "..."
```

```
context = "Document"
```

### After (version 3):

```
version = 3

[commands.default]
run = "claude -p" # Update this to your preferred CLI command
timeout_seconds = 120

[actions.rewrite]
type = "transform"
title = "Rewrite"
command = "default"
input_template = "..."
```

### Debug Mode

For the LSP server, set the IWE\_DEBUG environment variable. In debug mode, IWE LSP will generate a detailed log file named iwe.log in the directory where you started it:

```
export IWE_DEBUG=true; nvim
```

### When to Use Debug Mode

Including debug logs with your issue report will help us resolve problems faster. Debug mode is useful when:

- Troubleshooting CLI command behavior
- Understanding document processing steps
- Diagnosing LSP server issues
- Reporting bugs or unexpected behavior

Note: use -v 2 argument to see debug logs from CLI command.

### How to Use in Command Line

IWE provides a powerful command-line interface for managing markdown-based knowledge graphs. The CLI enables you to initialize projects, normalize documents, explore connections, export visualizations, and create consolidated documents.

### Quick Start

1. **Initialize a project:** iwe init
2. **Create a new document:** iwe new "My Note"
3. **Retrieve a document with context:** iwe retrieve -k my-note
4. **Find and search documents:** iwe find "search term"
5. **Normalize all documents:** iwe normalize
6. **View document hierarchy:** iwe tree
7. **Analyze your knowledge base:** iwe stats
8. **Export graph visualization:** iwe export dot
9. **Rename a document:** iwe rename old-key new-key
10. **Delete a document:** iwe delete document-key
11. **Extract a section:** iwe extract document --section "Title"
12. **Inline a reference:** iwe inline document --reference "other-doc"

### Installation & Setup

Before using the CLI, ensure IWE is installed and available in your PATH. Initialize any directory as an IWE project:

```
cd your-notes-directory
iwe init
```

This creates a `.iwe/` directory with configuration files.

## Global Usage

```
iwe [OPTIONS] <COMMAND>
```

## Global Options

- `-V, --version`: Display version information
- `-v, --verbose <LEVEL>`: Set verbosity level (default: 0)
  - 1: Minimal output (INFO level messages to stderr)
  - 2 or higher: Debug-level information to stderr
- `-h, --help`: Show help information

## Configuration

Commands respect settings in `.iwe/config.toml`:

```
[library]
path = "" # Subdirectory containing markdown files
```

```
[markdown]
normalize_headers = true
normalize_lists = true
```

## Command Categories

### Document Management

Command	Description	Documentation
<code>init</code>	Initialize a new IWE project	IWE Init
<code>new</code>	Create a new document	IWE New
<code>normalize</code>	Normalize all documents	IWE Normalize

### Document Retrieval

Command	Description	Documentation
<code>retrieve</code>	Retrieve document with context	IWE Retrieve
<code>find</code>	Search and discover documents	IWE Find
<code>tree</code>	Display document hierarchy	IWE Tree

### Refactoring Operations

Command	Description	Documentation
<code>rename</code>	Rename a document and update references	IWE Rename
<code>delete</code>	Delete a document and clean up references	IWE Delete
<code>extract</code>	Extract a section to a new document	IWE Extract
<code>inline</code>	Inline a referenced document	IWE Inline

### Analysis & Export

Command	Description	Documentation
<code>stats</code>	Analyze knowledge base statistics	IWE Stats



Command	Description	Documentation
export	Export graph visualization	IWE Export
squash	Squash documents	IWE Squash

## Exit Codes

Code	Meaning
0	Success - command completed without errors
1	Error - invalid arguments, missing files, operation failed

All commands return exit code 0 on success. On error, commands print a message to stderr and return exit code 1.

```
# Check exit code
iwe find "nonexistent-query"
echo $? # Returns 0 (empty result is not an error)

iwe retrieve --key "missing-doc"
echo $? # Returns 1 (document not found is an error)
```

## Concepts

### Inclusion Links

### Why Hierarchy Matters

Hierarchy is one of the most natural ways humans organize knowledge. We instinctively break complex information into nested structures:

- **Books** have chapters, sections, and paragraphs
- **Organizations** have departments, teams, and roles
- **Biology** classifies life into kingdoms, phyla, species
- **Outlines** structure ideas from general to specific

This isn't arbitrary—hierarchical thinking is how we manage complexity. It lets us zoom out for overview or zoom in for detail. It provides context: knowing something is “under” a topic tells you what it relates to.

Good knowledge systems should support this natural way of thinking.

### The Problem

Traditional ways of organizing information don't match how knowledge actually works.

### Directories: One Place Only

File systems force everything into a single hierarchy. A document about “React Performance Optimization” must live in either /frontend/react/ or /performance/—it can't naturally exist in both.

This leads to:

- Arbitrary placement decisions
- Broken references when folders change
- Duplicates or “misc” folders when nothing fits

But knowledge isn't a single tree—it's interconnected.

## Tags: Flexible but Shallow

Tags allow multiple categories, but they lack structure:

- No order or priority
- No explanation of relationships
- No hierarchy
- No grouping within a category

A document tagged `#react` `#performance` `#frontend` doesn't explain how these topics relate or which one is primary.

## The Solution: Inclusion Links

An **inclusion link** is a markdown link placed on its own line:

**# Frontend Development**

[React Fundamentals]([react-fundamentals.md](#))

[Vue.js Guide]([vue-guide.md](#))

[Performance Optimization]([performance.md](#))

When a link appears on its own line, it defines structure: “Frontend Development” becomes the parent of the linked documents.

This simple rule turns plain markdown into a structured, navigable system.

## What This Enables

### Multiple Contexts (Polyhierarchy)

A document can belong to multiple parents:

**# React Topics**

**# Performance Topics**

[Performance Optimization]

[Performance Optimization]

The same document appears in both contexts without duplication.

### Structure with Meaning

Unlike tags, inclusion links allow ordering, grouping, and explanation:

**# Projects**

**## Active**

[Website Redesign]([website-redesign.md](#))

Building the new company site

[Analytics Dashboard]([analytics.md](#))

Real-time metrics visualization

**## On Hold**

[Mobile App]([mobile-app.md](#))

Waiting for API completion

You're not just grouping items—you're adding context.

## Navigable Hierarchies

By linking documents together, you naturally create paths through your knowledge:

Knowledge Base > Work Projects > Website Redesign

Knowledge Base > Learning Notes > JavaScript Frameworks

This makes navigation and search more meaningful—you see not just what matches, but where it fits.

## Flexible Level of Detail

You can control how much information is visible:

- **Extract** sections into separate documents to hide details
- **Inline** documents to expand and show full content

Your knowledge base can expand or collapse depending on your needs.

## Inclusion Links vs Inline Links

### Inclusion Links (Structure)

Standalone links define parent-child relationships and are used for:

- Navigation
- Hierarchical traversal (--depth)
- Structured views

### Inline Links (References)

Links inside text create conceptual connections:

#### # Habit Formation

The [Habit Loop](habit-loop.md) consists of cue, routine, and reward.

This process is driven by [Dopamine Pathways](dopamine.md) in the brain.

These links:

- Create backlinks
- Show relationships between ideas
- Do not affect structure

## Why This Distinction Matters

When retrieving content with depth:

`!we retrieve psychology --depth 2`

- Inclusion links expand into full content
- Inline links remain references only

This keeps structure clean while preserving connections between ideas.

## Summary

Inclusion links turn markdown into a structured system without losing flexibility:

- No forced single hierarchy
- No flat, contextless tagging
- Documents can exist in multiple contexts
- Structure, ordering, and meaning are explicit
- Navigation and search become contextual

Instead of choosing where something belongs, you define how it connects.

## Features

### Notes Search

Notes search is a key feature in IWE. IWE allows you to organize documents hierarchy just by adding **Inclusion Links**. Then you can search for the documents taking into account the hierarchy.

Search can be used via the LSP Workspace Symbols command.

For every note, IWE will generate full paths. And allow you to do a fuzzy matching to filter the search results. So you can find both entries just by typing cappu.

Journal, 2025      ⇒   Week 3 - Coffee week   ⇒   Jan 26, 2025 - Cappuccino

My Coffee Journey   ⇒   Week 3 - Coffee week   ⇒   Jan 26, 2025 - Cappuccino

Since Week 3 is included in two notes it shown in both contexts.

Note that you don't have to deal with the file names at all, as everything is based on the headers from your notes!

### Custom Document Titles

By default, IWE uses the first header of each document as its title in search results. You can configure IWE to use a YAML frontmatter field instead by setting `frontmatter_document_title` in your configuration. See the Configuration documentation for details.

### Extract Actions

Extract actions enable the creation of new documents from markdown sections (headers). IWE provides two types of extract actions:

1. **Extract** - Extracts a single section into a new file
2. **Extract All** - Extracts all direct subsections of a section into separate files

Both operations:

- Create new files containing the selected content
- Add Inclusion Links (like [Section Title](new-file)) to the newly created files
- Automatically adjust header levels to maintain proper document structure
- Support relative path preservation

The reverse operation, known as **inline**, allows you to:

1. Embed the content back into the document via the inclusion link
2. Remove the link and the extracted file

### Extract Single Section

### Basic Configuration

The extract action is configured in `.iwe/config.toml` under the `[actions]` section:

```
[actions]
extract = { type = "extract", title = "Extract section", key_template = "{{id}}",
link_type = "markdown" }
```

### Configuration Options

- **type**: Must be "extract" for single section extraction
- **title**: Display name shown in editor's code actions menu

- **key\_template:** Template for generating the new file's key/name (see Template Variables below)
- **link\_type:** Optional link format - "markdown" for [text](key) or "wiki" for [[key]]

### Example: Basic Section Extraction

Source document (document.md):

# Main Document

## Important Section

This content will be extracted.

### Subsection

More content here.

After extraction (document.md):

# Main Document

[Important Section](extracted-123)

Extracted file (extracted-123.md):

# Important Section

This content will be extracted.

## Subsection

More content here.

### Extract All Subsections

#### Configuration

```
[actions]
extract_all = { type = "extract_all", title = "Extract all subsections", key_template = "{{title}}", link_type = "markdown" }
```

#### Configuration Options

- **type:** Must be "extract\_all" for extracting all subsections
- **title:** Display name shown in editor's code actions menu
- **key\_template:** Template for generating new file keys/names
- **link\_type:** Optional link format - "markdown" for [text](key) or "wiki" for [[key]]

### Example: Extract All Subsections

Source document (guide.md):

# User Guide

Introduction content here.

## Installation

How to install the software.

## Configuration

How to configure settings.

## ## Usage

How to use the application.

**After extract all** (guide.md):

## # User Guide

Introduction content here.

[Installation](Installation)

[Configuration](Configuration)

[Usage](Usage)

## Extracted files:

Installation.md:

## # Installation

How to install the software.

Configuration.md:

## # Configuration

How to configure settings.

Usage.md:

## # Usage

How to use the application.

## Template Variables

The key\_template supports several template variables for flexible file naming:

### Basic Variables

- **{{id}}**: Random unique identifier (e.g., 123, 456)
- **{{today}}**: Current date formatted using date\_format from [library] section (default: %Y-%m-%d)
- **{{title}}**: The section title being extracted (automatically sanitized for filenames)
- **{{slug}}**: URL-friendly version of the title (lowercase, alphanumeric characters only, non-alphanumeric replaced with dashes, no consecutive dashes)

### Parent Section Variables

- **{{parent.title}}**: Title of the parent section containing the extracted section
- **{{parent.slug}}**: URL-friendly version of the parent section title (lowercase, alphanumeric characters only, non-alphanumeric replaced with dashes, no consecutive dashes)
- **{{parent.key}}**: Key of the parent document

### Source Document Variables

- **{{source.key}}**: Full key of the source document
- **{{source.file}}**: Filename portion of the source document key
- **{{source.title}}**: Title (first header) of the source document

- `{{source.slug}}`: URL-friendly version of the source document title (lowercase, alphanumeric characters only, non-alphanumeric replaced with dashes, no consecutive dashes)
- `{{source.path}}`: Directory path of the source document

### Relative Path Behavior

IWE preserves directory structure when extracting sections. The key generation follows these rules:

1. **Generated keys are relative to the source document's directory**
2. **`Key::combine(&key.parent(), &relative_key)` creates the final path**
3. **The parent directory is automatically preserved**

### Example: Relative Path Extraction

Source document (docs/tutorial/basics.md):

```
# Basic Tutorial
```

```
## Getting Started
```

Content to extract.

#### Configuration:

```
extract = { type = "extract", title = "Extract", key_template = "extracted-{{title}}" }
```

#### Result:

- Source: docs/tutorial/basics.md
- Extracted file: docs/tutorial/extracted-Getting Started.md
- Link: [Getting Started](extracted-Getting Started)

The extracted file is created in the same directory (docs/tutorial/) as the source document.

### Advanced Configuration Examples

#### 1. Simple Numeric Keys

```
extract = { type = "extract", title = "Extract", key_template = "{{id}}" }
```

Creates files like: 123.md, 456.md

#### 2. Date-based Extraction

```
extract = { type = "extract", title = "Extract to today", key_template = "{{today}}" }
```

Creates files like: 2024-01-15.md

#### 3. Title-based Extraction

```
extract = { type = "extract", title = "Extract section", key_template = "{{title}}" }
```

Creates files like: Getting Started.md, Configuration.md

#### 4. Slug-based Extraction (URL-friendly)

```
extract = { type = "extract", title = "Extract as slug", key_template = "{{slug}}" }
```

Creates files like: getting-started.md, configuration.md From titles like “Getting Started”, “User’s Guide/Setup”, “API\*Reference” → getting-started.md, users-guide-setup.md, api-reference.md

## 5. Hierarchical Extraction

```
extract = { type = "extract", title = "Extract with context", key_template =
"{{parent.title}}-{{title}}" }
```

From a document with structure:

```
# User Guide
## Installation
```

Creates: User Guide-Installation.md

## 6. Hierarchical Extraction (URL-friendly)

```
extract = { type = "extract", title = "Extract with slug context", key_template =
"{{parent.slug}}-{{slug}}" }
```

From a document with structure:

```
# User Guide & Setup
## Installation/Configuration
```

Creates: user-guide-setup-installation-configuration.md

## 7. Source-aware Extraction

```
extract = { type = "extract", title = "Extract from source", key_template =
"{{source.file}}-{{title}}" }
```

From user-guide.md:

```
## Installation
```

Creates: user-guide-Installation.md

## 8. Source-aware Extraction (URL-friendly)

```
extract = { type = "extract", title = "Extract with source slug", key_template =
"{{source.slug}}-{{slug}}" }
```

From User Guide & Manual.md:

```
# User Guide & Manual
## Installation/Setup
```

Creates: user-guide-manual-installation-setup.md

## 9. Path-based Organization

```
extract = { type = "extract", title = "Extract to subfolder", key_template =
"extracted/{{title}}" }
```

From docs/guide.md, creates: docs/extracted/Installation.md

## 10. Wiki-style Links

```
extract = { type = "extract", title = "Extract (wiki)", key_template = "{{id}}",
link_type = "wiki" }
```

Creates links like: [[123]] instead of [Section Title](123)

## 11. Complex Template Example

```
extract = { type = "extract", title = "Extract with full context", key_template =
"{{source.path}}/{{today}}-{{parent.title}}-{{title}}" }
```

From docs/tutorials/advanced.md on 2024-01-15:



## # Advanced Tutorial

### ## Complex Features

Creates: docs/tutorials/2024-01-15-Advanced Tutorial-Complex Features.md

### Key Collision Handling

When the generated key already exists, IWE automatically appends numeric suffixes:

1. First attempt: extracted-section.md
2. If exists: extracted-section-1.md
3. If exists: extracted-section-2.md
4. And so on...

### Example: Handling Collisions

Multiple sections with same title:

#### # Document

##### ## Section

Content 1

##### ## Section

Content 2

##### ## Section

Content 3

With `key_template = "{{title}}"`, extract all creates:

- Section.md (first section)
- Section-1.md (second section)
- Section-2.md (third section)

### Date Formatting

The `{{today}}` variable uses the `date_format` setting from the `[library]` section:

```
[library]
```

```
date_format = "%Y-%m-%d" # Results in: 2024-01-15
```

```
[actions]
```

```
extract = { type = "extract", title = "Daily extract", key_template = "{{today}}-{{title}}" }
```

Common date formats:

- `%Y-%m-%d`: 2024-01-15
- `%Y%m%d`: 20240115
- `%b %d, %Y`: Jan 15, 2024
- `%A, %B %d, %Y`: Monday, January 15, 2024

### Filename Sanitization

Special characters in titles are automatically sanitized using the `sanitize_filename` crate:

- Section/With\*Special:Chars → SectionWithSpecialChars
- "Quoted Section" → Quoted Section
- Section | With | Pipes → Section With Pipes

## Command Line Usage

You can also extract sections using the CLI:

```
# List all sections with block numbers
iwe extract my-document --list

# Extract section by title
iwe extract my-document --section "Architecture"

# Extract section by block number
iwe extract my-document --block 2

# Preview changes
iwe extract my-document --section "Design" --dry-run
```

See IWE Extract for full documentation.

## Inline Notes

The inline action is a powerful feature for embedding content from one document directly into another. It replaces a link to a document with the actual content of that document. This is useful for consolidating notes, embedding sources, or restructuring your knowledge base.

There are two primary ways to inline content, configured via the `inline_type` property:

1. **section**: Inlines the entire content of the linked document into the current document's section that contains the link.
2. **quote**: Replaces the link with the content of the linked document, formatted as a markdown blockquote.

## Configuration

You can define custom inline actions in your `config.toml` file within the `.iwe` directory.

### Example Configuration:

Here is an example of how to configure two different inline actions: one for inlining as a section and another for inlining as a quote.

```
[actions]

# Inlines the content of the linked document and deletes the original file.
inline_section = { type = "inline", title = "Inline Section", inline_type = "section", keep_target = false }

# Inlines the content as a blockquote and keeps the original file.
inline_quote = { type = "inline", title = "Inline as Quote", inline_type = "quote", keep_target = true }
```

### Configuration Keys:

- `type`: Must be set to "inline".
- `title`: The text that will appear in the code action menu in your editor (e.g., "Inline Section").
- `inline_type`: Determines how the content is embedded.
  - "section": Embeds the full content.
  - "quote": Wraps the content in a blockquote.
- `keep_target` (optional, defaults to `false`):
  - If `false`, the original source file of the link will be deleted after its content is inlined. The action will also clean up any other references to the deleted file across your workspace.

- If true, the original source file is left untouched.

## Command Line Usage

You can also inline references using the CLI:

```
# List all [inclusion links](inclusion-links.md) with numbers
iwe inline my-document --list
```

```
# Inline by reference key
iwe inline my-document --reference "architecture"
```

```
# Inline by block number
iwe inline my-document --block 1
```

```
# Inline as blockquote
iwe inline my-document --reference "notes" --as-quote
```

```
# Keep the target document
iwe inline my-document --block 2 --keep-target
```

```
# Preview changes
iwe inline my-document --reference "design" --dry-run
```

See IWE Inline for full documentation.

## Linking and Note Templates

The “attach” code action allows you to link content under your cursor to another document. This feature is perfect for creating daily notes, collecting ideas in an inbox, or organizing thoughts into topic-specific documents.

Note: Enable Inlay Hints to see the action results immediately. (the list of backlinks will be updated automatically)

## How It Works

When you place your cursor on an Inclusion Links and trigger a code action, IWE can:

1. **Create or open** a target file based on your configuration
2. **Append** inclusion link under cursor to the target file

The target file is determined by templates you configure, allowing for dynamic file creation based on dates or static collection files.

## Common Use Cases

### Daily Note Collection

Automatically link interesting thoughts, tasks, or notes to today’s daily note:

```
[actions.daily]
type = "attach"
title = "Add to Daily Note"
key_template = "daily/{{today}}"
document_template = "# Daily Note - {{today}}\n\n{{content}}\n\n"
```

### Workflow:

1. While reading or writing, place cursor on any content
2. Trigger code action and select “Add to Daily Note”
3. Content gets appended to daily/2024-01-15.md (or created if doesn’t exist)

4. Continue your work knowing the important bits are captured

## Inbox System

Create an inbox for collecting random thoughts and ideas:

```
[actions.inbox]
type = "attach"
title = "Send to Inbox"
key_template = "inbox"
document_template = "# Inbox\n\n{{content}}\n\n"
```

### Workflow:

1. Come across something interesting while working
2. Place cursor on the content and select "Send to Inbox"
3. All collected items accumulate in a single `inbox.md` file
4. Review and organize your inbox regularly

## Topic Collections

Organize content by themes or projects:

```
[actions.research]
type = "attach"
title = "Add to Research Notes"
key_template = "research/general"
document_template = "# Research Notes\n\n{{content}}\n\n"
```

```
[actions.meeting_notes]
type = "attach"
title = "Add to Meeting Notes"
key_template = "meetings/{{today}}"
document_template = "# Meeting Notes - {{today}}\n\n{{content}}\n\n"
```

## File Creation Behavior

- **New files:** If the target file doesn't exist, it's created with the `document_template` content
- **Existing files:** Content is appended to the end of existing files
- **Duplicate prevention:** If the exact same inclusion link already exists in the target file, the code action will not be suggested

## Template Variables

Attach actions support two template variables:

- `{{today}}`: Current date formatted using your configured date format
  - In `key_template`: Uses `library.date_format` (default: "%Y-%m-%d")
  - In `document_template`: Uses `markdown.date_format` (default: "%b %d, %Y")
- `{{content}}`: The actual content being attached

## Editor Integration

The attach code action appears in your editor's code action menu (usually triggered by `Ctrl+.` or `Cmd+.`) when:

1. Your cursor is positioned on an inclusion link
2. You have attach actions configured in your `.iwe/config.toml`

## Best Practices

### 1. Start Simple

Begin with a basic daily note or inbox system:

```
[actions.capture]
type = "attach"
title = "Quick Capture"
key_template = "capture"
document_template = "# Capture\n\n{{content}}\n"
```

### 2. Use Descriptive Titles

Make it clear what each action does:

```
title = "Add to Today's Notes"    # Good
title = "Attach"                 # Less clear
```

### 3. Consider File Organization

Use path separators in key templates to organize files:

```
key_template = "daily/{{today}}"    # Creates files in daily/ folder
key_template = "projects/{{today}}" # Creates files in projects/ folder
```

## Integration with PKM Workflows

The attach action works well with common Personal Knowledge Management approaches:

- **GTD (Getting Things Done):** Use inbox actions for quick capture
- **Zettelkasten:** Create topic-specific collections and daily notes
- **PARA Method:** Organize attachments into Projects, Areas, Resources, Archives folders
- **Daily Notes:** Build a consistent journaling practice with date-based collection

## Creating Links from Text

The “link” code action allows you to quickly convert text under your cursor or selected text into a link to a new note. This feature streamlines the process of creating new notes while maintaining connections in your knowledge graph.

### How It Works

When you place your cursor on a word or select text and trigger a code action, IWE can:

1. **Create a new note** based on the text at your cursor or selection
2. **Replace the text** with a link to the newly created note
3. **Use templates** to control where the new note is created and how it's named

The link action works on a single line - you can either place your cursor on a word (and IWE will detect the word boundaries) or select a specific range of text.

## Configuration

Configure link actions in your `.iwe/config.toml`:

```
[actions.link]
type = "link"
title = "Link word"
key_template = "{{id}}"          # Template for the new note's key
link_type = "Markdown"          # Optional: "Markdown" or "WikiLink"
```

### Configuration Options

- **title:** The name displayed in your editor's code action menu

- **key\_template**: Template for generating the new note's key (supports template variables)
- **link\_type**: Optional link format
  - "Markdown": Creates [text] (key) style links (default)
  - "WikiLink": Creates [[key]] style links

## Common Use Cases

### Simple ID-Based Links

Create notes with auto-generated numeric IDs:

```
[actions.link]
type = "link"
title = "Link word"
key_template = "{{id}}"
```

#### Workflow:

1. Place cursor on “important” or select “important concept”
2. Trigger code action and select “Link word”
3. Text becomes [important] (2) and a new note 2.md is created with # important
4. Or if you selected “important concept”, it becomes [important concept] (2) with # important concept

### Slug-Based Links

Use slugified versions of the text as the note key:

```
[actions.link]
type = "link"
title = "Link word"
key_template = "{{slug}}"
```

#### Example:

- Cursor on “Important Concept” → [Important Concept] (important-concept) → important-concept.md
- Selected “My Idea” → [My Idea] (my-idea) → my-idea.md

### Title-Based Links

Use the exact text (sanitized for filenames) as the key:

```
[actions.link]
type = "link"
title = "Link word"
key_template = "{{title}}"
```

#### Example:

- Cursor on “MyWord” → [MyWord] (MyWord) → MyWord.md
- Selected “Project Ideas” → [Project Ideas] (Project Ideas) → Project Ideas.md

### WikiLink Format

Use WikiLink style instead of Markdown:

```
[actions.wiki_link]
type = "link"
title = "Link word (wiki)"
key_template = "{{id}}"
link_type = "WikiLink"
```

### Example:

- Cursor on “concept” → `[[2]]` → `2.md`
- Selected “important idea” → `[[2]]` → `2.md` with `# important idea`

### Text Selection vs Cursor

The link action supports two modes:

#### Cursor Mode (Word Detection)

- Place your cursor anywhere within a word
- IWE automatically detects word boundaries
- Words can include alphanumeric characters, underscores, hyphens, and Unicode characters
- Examples:
  - Cursor on “wo|rd” → detects “word”
  - Cursor on “multi-w|ord” → detects “multi-word”
  - Cursor on “some\_|function” → detects “some\_function”

#### Selection Mode

- Select any text range on a single line
- IWE uses exactly what you selected
- Perfect for phrases with spaces or partial words
- Examples:
  - Select “important concept” → creates link for “important concept”
  - Select “very important” → creates link for “very important”
  - Select part of a sentence → creates link for that specific text

### Template Variables

Link actions support several template variables:

- `{{id}}`: Auto-generated unique numeric ID
- `{{slug}}`: Slugified version of the text (lowercase, hyphens instead of spaces)
- `{{title}}`: Sanitized version of the text (safe for filenames)
- `{{today}}`: Current date formatted using your configured date format from `library.date_format` (default: “%Y-%m-%d”)

### Key Collision Handling

If a note with the generated key already exists, IWE automatically appends a numeric suffix:

#### Example:

- First link: `concept` → `concept.md`
- Second link: `concept` → `concept-1.md`
- Third link: `concept` → `concept-2.md`

### Use Multiple Link Actions

Configure different link actions for different purposes:

```
[actions.link]
type = "link"
title = "Quick Link"
key_template = "{{id}}"
```

```
[actions.concept_link]
type = "link"
title = "Link Concept"
```

```
key_template = "concepts/{{slug}}"
```

```
[actions.daily_link]  
type = "link"  
title = "Link to Today"  
key_template = "daily/{{today}}"
```

## Delete Action

The delete action allows you to cleanly remove a referenced section and automatically update all files that reference it.

## How It Works

When you place your cursor on an Inclusion Links (like [Important Topic](file)) and trigger the delete action, IWE will:

1. **Delete the target file** - The referenced section/file is completely removed
2. **Clean up inclusion links** - All inclusion links to the deleted section are removed from other files
3. **Convert inline links** - Inline links to the deleted section are converted to plain text, preserving readability

## Usage

1. Position your cursor on any inclusion link in your markdown file
2. Open the code actions menu (typically Ctrl+. or Cmd+.)
3. Select "Delete" from the refactor actions

## Example

### Before deletion:

```
# My Notes
```

Some text with an inline link to [Important Topic](file).

```
[Important Topic](file)
```

### After deleting the reference on line with [Important Topic](5):

```
# My Notes
```

Some text with an inline link to Important Topic.

The referenced file Important Topic is completely deleted, the inclusion link is removed, and the inline link becomes plain text.

## When Delete Action Is Available

- The delete action only appears when your cursor is on an **inclusion link**
- It will not appear on regular text, headers, or other content types
- The action ensures safe deletion by updating all referencing files automatically

## Command Line Usage

You can also delete documents using the CLI:

```
# Delete with confirmation prompt  
iwe delete my-document
```

```
# Preview changes first
```



```
iwe delete my-document --dry-run
```

```
# Skip confirmation
```

```
iwe delete my-document --force
```

```
# Get affected document keys
```

```
iwe delete my-document --keys
```

See IWE Delete for full documentation.

## Navigation

IWE provides several ways to navigate your documents using standard LSP features. This allows you to move through your knowledge graph efficiently using familiar editor commands.

### Links Navigation (Go To Definition)

Jump directly to linked documents using the LSP “Go To Definition” command.

#### Usage

1. Place your cursor on a markdown link like [Topic](topic-file)
2. Trigger Go To Definition:
  - **VS Code:** F12 or Ctrl+Click / Cmd+Click
  - **Neovim:** gd or :lua vim.lsp.buf.definition()
  - **Helix:** gd
3. The linked document opens automatically

#### Example

```
# My Notes
```

Check out [Project Ideas](project-ideas) for brainstorming.

With your cursor on “Project Ideas”, triggering Go To Definition opens project-ideas.md.

### Table of Contents (Document Symbols)

View the structure of your current document using the LSP “Document Symbols” command.

#### Usage

1. Open any markdown file
2. Trigger Document Symbols:
  - **VS Code:** Ctrl+Shift+0 / Cmd+Shift+0
  - **Neovim:** :lua vim.lsp.buf.document\_symbol() or use Telescope
  - **Helix:** space + s
3. Navigate through headers and sections

This provides an outline view of your document, showing all headers in a hierarchical tree. You can quickly jump to any section.

### Backlinks (Find References)

Find all documents that link to the current document using the LSP “Find References” command.

#### Usage

1. Open a document you want to find references to
2. Trigger Find References:
  - **VS Code:** Shift+F12 or right-click and select “Find All References”
  - **Neovim:** :lua vim.lsp.buf.references() or gr

- **Helix:** `gr`

3. View all documents containing links to this file

### Example

If you're viewing `project-ideas.md` and trigger Find References, you'll see a list of all documents that contain links like `[Project Ideas](project-ideas)`.

This is essential for understanding how your notes are connected and for discovering relationships in your knowledge graph.

### Navigation Tips

- **Preview before jumping:** Use Hover Preview to see linked content without leaving your current document
- **Use search for discovery:** Combine navigation with Notes Search to find documents by content path
- **Track your trail:** Some editors maintain a navigation history, allowing you to go back to previous locations

### Hover Preview

IWE supports the standard LSP `textDocument/hover` request to preview the contents of a linked note without navigating away from the current document.

### Supported links

- Wiki links: `[[note]]`
- Markdown links: `[title](note)`

External links (e.g. `https://...`, `mailto:...`) are ignored.

### Preview content

- Returns the target note as Markdown (MarkupContent with kind `markdown`).
- Strips frontmatter at the top of the note (delimited by `---` and terminated by `---` or `...`).
- Returns the rest of the document without truncation so your editor can decide how to render/clip it.

### Editor usage

- Helix: place the cursor on a link, then `space + k`
- Neovim: place the cursor on a link, then `K` (or run `:lua vim.lsp.buf.hover()`)
- VS Code: hover the link (or run "Show Hover")

### Text Structure Normalization / Formatting

IWE provides auto-formatting through the LSP formatting feature. This typically triggers when you save your document (if your editor is configured for format-on-save) or manually via a formatting command.

### What Gets Normalized

#### 1. Link Title Updates

Link titles are automatically updated to match the header of the linked document.

#### Before:

See `[old title](my-note)` for details.

**After** (if `my-note.md` has header `"# My Updated Note"`):

See [My Updated Note]([my-note](#)) for details.

## 2. Header Level Adjustment

Header levels are adjusted to maintain a proper tree structure. This ensures that nested sections have correct relative header levels.

**Before:**

```
# Main Document
```

```
#### Incorrectly Deep Section
```

Some content.

**After:**

```
# Main Document
```

```
## Incorrectly Deep Section
```

Some content.

## 3. Ordered List Numbering

Ordered lists are renumbered sequentially, regardless of the original numbers.

**Before:**

1. First item
5. Second item
3. Third item

**After:**

1. First item
2. Second item
3. Third item

## 4. List Indentation and Spacing

List structure is normalized with consistent indentation and proper newlines.

**Before:**

- Item one
  - Badly indented
  - Inconsistent
- No space after marker

**After:**

- Item one
  - Badly indented
  - Inconsistent
- No space after marker

## 5. Whitespace Cleanup

Excess blank lines and trailing whitespace are cleaned up.

## Usage

### Format on Save

Most editors can be configured to format on save:

- **VS Code:** Enable “Format On Save” in settings
- **Neovim:** Configure autocommand or use plugin like `conform.nvim`
- **Helix:** Set `auto-format = true` in `languages.toml`

### Manual Formatting

Trigger formatting manually:

- **VS Code:** `Shift+Alt+F` / `Shift+Option+F`
- **Neovim:** `:lua vim.lsp.buf.format()`
- **Helix:** `:format`

### Configuration

Formatting behavior can be configured in `.iwe/config.toml`:

```
[markdown]
normalize_headers = true # Adjust header levels
normalize_lists = true   # Fix list formatting
```

### Tips

- **Enable format-on-save** for consistent formatting across your library
- **Use with version control** to easily review formatting changes
- If you notice unexpected formatting, check Header Levels Normalization for detailed header adjustment rules

### Inlay Hints

Inlay hints are visual annotations that appear inline within your document, providing contextual information without requiring navigation. IWE provides two types of inlay hints.

#### Note Header Hints

At the top of each document, IWE displays information about the note’s position in your knowledge graph.

#### What’s Shown

- **Parent document title:** Shows which document links to this note
- **Links counter:** Number of incoming references (backlinks)

#### Example

When viewing `project-ideas.md`:

```
[linked from: Index] [3 links]
# Project Ideas
```

Your content here...

This tells you that “Index” links to this document and there are 3 total backlinks.

#### Inclusion Hints

When viewing an Inclusion Links, inlay hints show which notes directly reference the linked content.

## What's Shown

- **Parent notes list:** The direct parent notes (one level up) that link to the referenced content

## Example

### # Daily Notes

[Meeting Notes](meeting-2024) [Index, Journal]

The hint [Index, Journal] shows that the linked note is referenced from both “Index” and “Journal” documents.

## Enabling Inlay Hints

### VS Code

Inlay hints should work automatically. If not visible, check:

1. Open Settings (Ctrl+, / Cmd+,)
2. Search for “inlay hints”
3. Enable “Editor: Inlay Hints”

### Neovim

Enable inlay hints in your LSP configuration:

```
vim.lsp.inlay_hint.enable(true)
```

Or toggle with a keybinding:

```
vim.keymap.set('n', '<leader>ih', function()  
  vim.lsp.inlay_hint.enable(not vim.lsp.inlay_hint.is_enabled())  
end)
```

### Helix

Inlay hints are enabled by default. Configure in config.toml:

```
[editor.lsp]  
display-inlay-hints = true
```

## Benefits

- **Context at a glance:** Understand note relationships without navigating away
- **Track connections:** See how many documents link to the current note
- **Navigate hierarchy:** Understand where content fits in your knowledge structure

## Auto-Complete

IWE can suggest links as you type using the standard LSP code completion feature.

## Link Format

By default, completions insert Markdown-style links [title](key). You can configure IWE to use WikiLinks [[key]] instead:

```
[completion]  
link_format = "wiki"
```

Available options:

- "markdown" (default): Creates [title](key) style links
- "wiki": Creates [[key]] style WikiLinks

## Document Titles

By default, IWE uses the first header of a document as its title in completion suggestions. You can configure IWE to use a YAML frontmatter field instead:

```
[library]
frontmatter_document_title = "title"
```

With this configuration, documents with frontmatter like:

```
---
title: Custom Document Title
---
```

### # Header

Will appear in completions as “Custom Document Title” and insert [Custom Document Title](key) when selected. If the frontmatter field is missing, IWE falls back to using the first header.

## Text Manipulation

IWE offers actions for context-aware transformations on your notes. These actions are available through your editor’s code actions menu (usually Ctrl+. or Cmd+.).

## List to Sections

Convert a list into a series of headers/sections. Each list item becomes a section header with its nested content preserved.

### Example

#### Before:

### # Topics

- Project A
  - Details about project A
  - More info
- Project B
  - Details about project B

#### After:

### # Topics

#### ## Project A

Details about project A

More info

#### ## Project B

Details about project B

## When to Use

- Converting brainstorm lists into structured documents
- Expanding outline notes into full sections
- Promoting list items to top-level content

## Sections to List

Convert a series of headers back into a list. This is the reverse of “List to Sections”.

### Example

#### Before:

# Topics

## Project A

Details about project A

## Project B

Details about project B

#### After:

# Topics

- Project A
  - Details about project A
- Project B
  - Details about project B

## When to Use

- Condensing detailed sections into an overview
- Creating summary lists from expanded content
- Reorganizing document structure

## Change List Type

Toggle between bullet lists and ordered (numbered) lists.

### Example

#### Before (bullet list):

- First item
- Second item
- Third item

#### After (ordered list):

1. First item
2. Second item
3. Third item

## When to Use

- Converting unordered lists to numbered steps
- Changing numbered lists back to bullet points
- Adjusting list style based on content type

## Sort List Items

Sort list items alphabetically or by other criteria.

### Example

#### Before:

- Zebra
- Apple
- Mango
- Banana

#### After:

- Apple
- Banana
- Mango
- Zebra

#### When to Use

- Organizing alphabetical lists (glossaries, indexes)
- Sorting task lists
- Maintaining consistent ordering

#### Usage

1. Place your cursor on the list or section you want to transform
2. Open the code actions menu:
  - **VS Code:** Ctrl+. / Cmd+.
  - **Neovim:** :lua vim.lsp.buf.code\_action()
  - **Helix:** space + a
3. Select the desired transformation from the menu

#### Header Levels Normalization

IWE reads and understands nested structures based on headers. It identifies sub-header relationships. Markdown allows header structures where the nesting isn't clear, like:

```
## First Header
```

```
# Second Header
```

IWE automatically fixes the header levels to ensure they're nested correctly. So the example above corrects to:

```
# First Header
```

```
# Second Header
```

#### Removing unnecessary levels

IWE can normalize the headers structure by dropping unnecessary header levels, for example:

```
# First header
```

```
### Second header
```

Will be normalized by dropping unnecessary levels and will look like:

```
# First header
```

```
## Second header
```

#### Files Renaming

IWE provides file renaming through the LSP rename refactoring feature. When you rename a note file, IWE automatically updates all references throughout your entire library.



## How It Works

When you trigger a rename operation on a markdown file:

1. **Rename the file** - The file is renamed to your specified name
2. **Update all references** - Every link pointing to the old filename is updated to use the new name
3. **Preserve link titles** - Link display text remains unchanged

This ensures your knowledge graph stays consistent without manual search-and-replace operations.

## Usage

### In Your Editor

1. Open a markdown file you want to rename
2. Trigger the LSP rename command:
  - **VS Code**: F2 or right-click and select “Rename Symbol”
  - **Neovim**: `:lua vim.lsp.buf.rename()` or your configured keybinding
  - **Helix**: `space + r`
3. Enter the new filename
4. Confirm the rename

### Example

#### Before renaming:

old-topic.md:

# Old Topic

Some content here.

index.md:

# Index

See [Old Topic](old-topic) for details.

Check also [Old Topic](old-topic) in another context.

#### After renaming to new-topic.md:

new-topic.md:

# Old Topic

Some content here.

index.md:

# Index

See [Old Topic](new-topic) for details.

Check also [Old Topic](new-topic) in another context.

Note that the link text (“Old Topic”) is preserved while the link target is updated.

### Command Line Usage

You can also rename documents using the CLI:

```
# Basic rename
```

```
iwe rename old-topic new-topic
```

```
# Preview changes first
```

```
iwe rename old-topic new-topic --dry-run
```

```
# Get affected document keys
```

```
iwe rename old-topic new-topic --keys
```

See IWE Rename for full documentation.

### Benefits

- **Safe refactoring** - No broken links after renaming
- **Bulk updates** - All references updated in a single operation
- **Undo support** - Most editors support undoing the rename operation

### Graph Visualization

IWE provides powerful graph visualization capabilities through DOT format export, allowing you to create visual representations of your knowledge graph structure. This helps you understand the relationships between documents, sections, and references in your markdown collection.

### Export Command

The `iwe export dot` command generates graph data in DOT format, which can be processed by Graphviz and other visualization tools.

### Basic Usage

```
# Export all root documents
```

```
iwe export dot
```

```
# Export specific document by key
```

```
iwe export dot --key project-notes
```

```
# Export with depth limit
```

```
iwe export dot --depth 3
```

### Advanced Visualization with Headers

Use the `--include-headers` flag to create detailed visualizations that show document structure with sections grouped in colored subgraphs:

```
# Include sections and subgraphs
```

```
iwe export dot --include-headers
```

```
# Detailed view of specific document
```

```
iwe export dot --key documentation --include-headers
```

```
# Combined with depth limit
```

```
iwe export dot --key meetings --depth 2 --include-headers
```

### Visualization Modes

#### Basic Mode (Default)

Shows document-to-document relationships with clean node styling:

```
digraph G {
  rankdir=LR
  fontname=Verdana

  1[label="Project Notes",fillcolor="#ffeaea",fontsize=16,shape=note,style=filled]
  2[label="Meeting Notes",fillcolor="#f6e5ee",fontsize=16,shape=note,style=filled]
```

```
1 -> 2 [color="#38546c66",arrowhead=normal,penwidth=1.2]
}
```

### Detailed Mode (-include-headers)

Shows document structure with sections grouped in colored subgraphs:

```
digraph G {
    rankdir=LR

    1[label="Project Notes",shape=note,style=filled]
    2[label="Introduction",shape=plain]
    3[label="Requirements",shape=plain]

    subgraph cluster_0 {
        labeljust="l"
        style=filled
        color="#fff9de"
        fillcolor="#fff9de"
        2
        3
    }

    2 -> 1[arrowhead="empty",style="dashed"]
    3 -> 1[arrowhead="empty",style="dashed"]
}
```

### Key Features

- **Color Coding:** Each document key gets a unique, consistent color scheme
- **Shape Differentiation:** Documents use note shape, sections use plain shape
- **Subgraph Clustering:** Sections are grouped in colored clusters with document keys
- **Edge Styles:** Different styles for document vs section relationships
- **Automatic Layout:** Left-to-right layout optimized for readability

### Integration with Graphviz

#### Generate PNG Images

```
# Basic visualization
iwe export dot | dot -Tpng -o knowledge-graph.png

# Detailed with sections
iwe export dot --include-headers | dot -Tpng -o detailed-graph.png

# Focus on specific topic
iwe export dot --key project --include-headers | dot -Tpng -o project-structure.png
```

#### Generate SVG for Web

```
# Scalable vector graphics
iwe export dot | dot -Tsvg -o interactive-graph.svg

# With better layout for complex graphs
iwe export dot --include-headers | neato -Tsvg -o network-view.svg
```

### Different Layout Engines

```
# Hierarchical layout (default)
iwe export dot | dot -Tpng -o hierarchical.png
```

```
# Force-directed layout
iwe export dot | neato -Tpng -o network.png

# Circular layout
iwe export dot | circo -Tpng -o circular.png

# Spring-based layout
iwe export dot | fdp -Tpng -o spring.png
```

## Filtering and Focusing

### By Document Key

```
# Show only documents related to 'meetings'
iwe export dot --key meetings --include-headers

# Multiple levels of related documents
iwe export dot --key architecture --depth 2
```

### By Content Depth

```
# Show only immediate relationships
iwe export dot --depth 1

# Show deeper connections
iwe export dot --depth 3 --include-headers
```

## Workflow Examples

### Daily Documentation Review

```
#!/bin/bash
# Generate today's knowledge graph
iwe export dot --include-headers > today.dot
dot -Tpng today.dot -o daily-review.png
open daily-review.png # macOS
```

### Project Structure Analysis

```
#!/bin/bash
# Analyze specific project structure
iwe export dot --key $PROJECT_NAME --include-headers | \
  dot -Tsvg -o "project-${PROJECT_NAME}.svg"
```

### Knowledge Base Overview

```
#!/bin/bash
# Create multiple views of your knowledge base
iwe export dot > overview.dot
iwe export dot --include-headers > detailed.dot

# Generate both views
dot -Tpng overview.dot -o overview.png
dot -Tpng detailed.dot -o detailed.png
```

## Customization Tips

### Layout Optimization

For large graphs, experiment with different Graphviz engines:

- **dot**: Best for hierarchical structures
- **neato**: Good for network-like relationships

- **fdp**: Spring model, useful for clustered data
- **circo**: Circular layout for cyclic structures

## Output Formats

Graphviz supports many output formats:

- **PNG/JPG**: For presentations and documents
- **SVG**: For interactive web displays
- **PDF**: For high-quality prints
- **DOT**: For further processing or debugging

## Performance Considerations

- Use `--depth` limits for large knowledge bases
- Filter by `--key` to focus on specific areas
- Use `--include-headers` for detailed structure visualization when needed

## Troubleshooting

### Large Graphs

# Reduce complexity with depth limits

```
iwe export dot --depth 2 | dot -Tpng -o simplified.png
```

# Use different layout engine

```
iwe export dot | fdp -Tpng -o alternative-layout.png
```

### Missing Graphviz

Install Graphviz on your system:

# macOS

```
brew install graphviz
```

# Ubuntu/Debian

```
sudo apt install graphviz
```

# Windows

```
winget install graphviz
```

### Complex Layouts

For complex graphs, try different approaches:

# Increase node separation

```
iwe export dot | dot -Tpng -Gnodesep=1.0 -o spaced.png
```

# Adjust DPI for clarity

```
iwe export dot | dot -Tpng -Gdpi=200 -o high-res.png
```

The visualization feature makes IWE's knowledge management capabilities tangible, helping you understand and navigate your documentation structure at a glance.

### Sub-Directories

IWE supports organizing your markdown files in subdirectories while maintaining full functionality across all features. This allows you to structure your knowledge base hierarchically without losing the ability to link, search, and process files across directory boundaries.

## How It Works

### Recursive Directory Scanning

IWE recursively scans the configured library path and all its subdirectories:

- **Includes:** All .md files in any subdirectory level
- **Excludes:** Hidden files and directories (starting with .)
- **File Keys:** Include the relative path from library root

### File Path Resolution

Files in subdirectories get keys that include their relative path:

Project Structure:

```
your-project/
├── .iwe/config.toml
├── docs/
│   ├── guide.md          → Key: "docs/guide"
│   ├── api/
│   │   ├── reference.md  → Key: "docs/api/reference"
│   │   └── examples/
│   │       └── basic.md   → Key: "docs/examples/basic"
└── README.md             → Key: "README" (if library.path = "")
```

### Cross-Directory Linking

Links use relative paths based on each file's location in the directory structure:

```
<!-- In index.md (root level) -->
```

See the [guide](docs/guide.md) for details.

```
<!-- In docs/guide.md -->
```

Back to [index](../index.md) or see [API reference](api/reference.md).

```
<!-- In docs/api/reference.md -->
```

Check out the [basic example](../examples/basic.md) or [guide](../guide.md).

### Path Resolution Rules:

- From root to subdirectory: subdirectory/file.md
- From subdirectory to root: ../file.md
- Between subdirectories at same level: ../other-directory/file.md
- Within same directory: file.md

### Custom Text Commands

IWE includes **text transformation** capabilities that can be accessed right from your text editor. You can effortlessly **rewrite** text, **expand** on ideas, **highlight** important words, or apply any custom transformation using CLI tools.

Transform actions work by piping content through external commands. You can use custom scripts, standard Unix tools, or AI assistants like `claude -p` (Claude CLI in pipe mode) - any command-line tool that reads from stdin and writes to stdout.

## How It Works

1. Select or place cursor on a text block in your editor
2. Open the code actions menu (Ctrl+. or Cmd+.)
3. Select a transform action (e.g., "Rewrite", "Expand")
4. The text is processed through your configured command

5. The result replaces the original content

## Configuration

Transform actions require two parts:

1. **Command definition** - the CLI tool to run
2. **Action definition** - how to use the command

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120

[actions.rewrite]
type = "transform"
title = "Rewrite"
command = "claude"
input_template = ""
Rewrite this text to improve clarity and readability.
Keep links intact.

{{context}}
"""
```

Check Configuration section for detailed instructions on how to set up custom commands.

## Example Commands

### Using Claude CLI:

```
[commands.claude]
run = "claude -p"
timeout_seconds = 120
```

### Using a Python script:

```
[commands.summarize]
run = "python ~/scripts/summarize.py"
timeout_seconds = 30
```

### Using standard Unix tools:

```
[commands.uppercase]
run = "tr '[:lower:]' '[:upper:]'"
timeout_seconds = 5
```

## Editor specifics

### Helix

#### Installation & Setup

First, the `iwes` binary needs to be available on your system `$PATH`. Please see the installation instructions and pick your preferred method of installation. I recommend the AUR for Arch users.

Next, you'll need to add `iwe` as an LSP and enable it for files.

#### Setup Snippet

```
# ` $HOME/.config/helix/languages.toml`

[language-server.iwe]
command = "iwes"
```

```
[[language]]
name = "markdown"
language-servers = ["iwe"]
# You can add other LSPs here, too:
# language-servers = ["iwe", "marksman"]

# NOTE: You may consider disabling
# autoformat if you're having issues
# with tables!
auto-format = true
```

## Setup IWE Only For Your Notes

You probably don't want iwe enabled for **every Markdown file you ever open**. For example, you may not want its features when you're working on README files for different projects. In that case, I recommend Helix's project-specific configuration feature. In the root of your notes directory, you can create a folder called `.helix`, add a file called `languages.toml` and put the setup snippet in there.

## Usage

Please refer to the usage guide for a quick reference.

## Hover Preview

To preview a wiki/markdown-linked note without navigating away, place your cursor on the link and trigger LSP hover (Helix default: `space + k`).

More details: [Hover Preview](#)

## Common Keybindings

Action	Keybinding
Go to definition (follow link)	<code>gd</code>
Find references (backlinks)	<code>gr</code>
Hover preview	<code>space + k</code>
Code actions	<code>space + a</code>
Document symbols (outline)	<code>space + s</code>
Workspace symbols (search)	<code>space + S</code>
Rename file	<code>space + r</code>
Format document	<code>: format</code>

## Code Actions

To use IWE code actions (extract, inline, attach, etc.):

1. Place cursor on the target content
2. Press `space + a` to open code actions menu
3. Select the desired action

## Helix-Specific Notes

- **Buffer management after inline:** When you inline a section (merge an extracted note back), the buffer for the deleted file remains open. Use `:buffer-close` or `:bc` to close it manually.
- **Auto-format with tables:** If you work with markdown tables, you may want to disable auto-format to prevent unwanted table reformatting:



```
[[language]]
name = "markdown"
language-servers = ["iwe"]
auto-format = false
```

- **Multiple language servers:** You can use IWE alongside other markdown language servers:

```
[[language]]
name = "markdown"
language-servers = ["iwe", "marksman"]
```

## VS Code

### Installation & Setup

#### Install the IWE Extension

The IWE extension is available on the Visual Studio Code Marketplace:

#### Option 1: Via VS Code Marketplace

1. Open VS Code
2. Go to Extensions view (Ctrl+Shift+X / Cmd+Shift+X)
3. Search for “IWE”
4. Click “Install” on the IWE extension

#### Option 2: Via Command Line

```
code --install-extension IWE.iwe
```

#### Option 3: Direct Link

Visit the IWE extension on VS Code Marketplace

#### Prerequisites

The IWE extension requires the `iwes` LSP server binary to be installed on your system:

1. **Install via Cargo** (recommended):

```
cargo install iwe
```

2. **Download from GitHub Releases:**

- Visit IWE releases
- Download the appropriate binary for your system
- Ensure `iwes` is in your system PATH

3. **Build from Source:**

```
git clone https://github.com/iwe-org/iwe.git
cd iwe
cargo build --release --bin iwes
# Copy target/release/iwes to your PATH
```

#### Verify Installation

1. Open VS Code in a directory with markdown files
2. Open a `.md` file
3. Check the bottom status bar - you should see “IWE” indicating the language server is active
4. Try using IWE features (see shortcuts below)

## VS Code Shortcuts for IWE Actions

### Core Actions

IWE Feature	VS Code Shortcut	Alternative Access
<b>Code Actions</b> (Extract/Inline/Transform)	Ctrl+. / Cmd+.	Right-click → “Quick Fix...”
<b>Go to Definition</b> (Follow Links)	F12	Right-click → “Go to Definition”
<b>Find All References</b> (Backlinks)	Shift+F12	Right-click → “Go to References”
<b>Document Symbols</b> (Table of Contents)	Ctrl+Shift+O / Cmd+Shift+O	Command Palette → “Go to Symbol”
<b>Workspace Search</b> (Global Search)	Ctrl+T / Cmd+T	Command Palette → “Go to Symbol in Workspace”
<b>Format Document</b> (Auto-Format)	Shift+Alt+F / Shift+Option+F	Right-click → “Format Document”
<b>Rename Symbol</b> (Rename File)	F2	Right-click → “Rename Symbol”

### Additional VS Code Features

Feature	Shortcut	Description
<b>Command Palette</b>	Ctrl+Shift+P / Cmd+Shift+P	Access all IWE commands
<b>Auto-Complete</b>	Ctrl+Space / Cmd+Space	Trigger link completion while typing
<b>Peek Definition</b>	Alt+F12 / Option+F12	Preview linked document without opening
<b>Peek References</b>	Shift+Alt+F12 / Shift+Option+F12	Preview backlinks without opening

### Command Palette Access

All IWE features are also available via the Command Palette (Ctrl+Shift+P / Cmd+Shift+P):

- Type “IWE” to see all available commands
- Type “Go to” for navigation commands
- Type “Format” for formatting commands
- Type “Rename” for refactoring commands

### Usage Examples

#### Extracting a Section

1. Place cursor on a header line (e.g., ## Section Title)
2. Press Ctrl+. / Cmd+. to open Quick Actions
3. Select “Extract section”
4. VS Code will create a new file and replace the section with a link

#### Following Links

1. Click on any markdown link or place cursor within brackets
2. Press F12 or Ctrl+Click / Cmd+Click
3. VS Code will navigate to the target document

## Finding Backlinks

1. Place cursor on a header or anywhere in a document
2. Press Shift+F12
3. VS Code will show all documents that link to the current location
4. Click any result to navigate

## Text Transform Actions

1. Select text you want to modify
2. Press Ctrl+. / Cmd+.
3. Choose from available transform actions (if configured)
4. The selected text will be processed and replaced

## Global Search

1. Press Ctrl+T / Cmd+T
2. Type search terms
3. VS Code will show matching documents and sections
4. Use arrow keys to navigate results, Enter to open

## Configuration

### Workspace Settings

Create or edit `.vscode/settings.json` in your workspace:

```
{
  "iwe.enable": true,
  "iwe.trace.server": "off",
  "files.associations": {
    "*.md": "markdown"
  },
  "markdown.validate.enabled": true
}
```

### User Settings

For global IWE configuration, edit your VS Code user settings:

1. Open Settings (Ctrl+, / Cmd+,)
2. Search for “IWE”
3. Configure available options

## Features in VS Code

### Auto-Complete

- **Link Completion:** Type [ and get suggestions for existing documents
- **Smart Suggestions:** Context-aware completions based on document structure
- **Snippet Support:** Quick insertion of common markdown patterns

### Visual Enhancements

- **Inlay Hints:** See parent document references and link counts
- **Syntax Highlighting:** Enhanced markdown highlighting with IWE-specific elements
- **Error Detection:** Real-time validation of links and structure

### File Management

- **Auto-Save:** New files created by extraction are automatically saved
- **File Watching:** Changes are tracked and processed in real-time

- **Project Integration:** Works with VS Code's built-in file explorer

## Troubleshooting

### Common Issues

#### 1. LSP Server Not Starting

- Check that iwe is installed and in PATH
- Restart VS Code
- Check Output panel → "IWE Language Server" for errors

#### 2. Features Not Working

- Ensure you're in a directory with `.iwe/config.toml`
- Verify the file is saved as `.md`
- Check VS Code status bar for IWE indicator

#### 3. Performance Issues

- Large workspaces may be slow; consider using library path configuration
- Disable unnecessary VS Code extensions
- Check system resources

## Debug Mode

Enable debug logging:

1. Set environment variable: `IWE_DEBUG=true`
2. Restart VS Code
3. Check the IWE log file in your workspace directory
4. Include logs when reporting issues

## Getting Help

- **GitHub Issues:** Report bugs or request features
- **Discussions:** Community support and questions
- **Documentation:** Full documentation wiki

## Best Practices for VS Code

1. **Use Workspace Folders:** Open your entire knowledge base as a workspace folder
2. **Configure File Associations:** Ensure all markdown files are properly associated
3. **Enable Auto-Save:** Prevent data loss with VS Code's auto-save feature
4. **Use Split Views:** Work with multiple documents simultaneously
5. **Organize with Explorer:** Use VS Code's file explorer alongside IWE's navigation
6. **Keyboard Shortcuts:** Learn the shortcuts for faster workflow
7. **Extensions Integration:** IWE works well with other markdown extensions

## Neovim

### Installation & Setup

#### Install the IWE Plugin

The IWE Neovim plugin is available at: `iwe.nvim`

#### Option 1: Using `lazy.nvim` (recommended)

```
{  
  "iwe-org/iwe.nvim",  
  dependencies = {  
    "nvim-lua/plenary.nvim", "nvim-telescope/telescope.nvim",  
  },  
}
```

```

config = function()
    require("iwe").setup()
end,
}

```

### Option 2: Using packer.nvim

```

use {
    "iwe-org/iwe.nvim",
    requires = {
        "nvim-lua/plenary.nvim",
        "nvim-telescope/telescope.nvim",
    },
    config = function()
        require("iwe").setup()
    end,
}

```

### Option 3: Using vim-plug

```

Plug 'nvim-lua/plenary.nvim'
Plug 'nvim-telescope/telescope.nvim'
Plug 'iwe-org/iwe.nvim'

" Add to your init.vim after plug#end()
lua require("iwe").setup()

```

### Option 4: Manual Installation

```

git clone https://github.com/iwe-org/iwe.nvim.git ~/.local/share/nvim/site/pack/
plugins/start/iwe.nvim

```

### Prerequisites

The IWE plugin requires the iwes LSP server binary to be installed on your system:

#### 1. Install via Cargo (recommended):

```
cargo install iwe
```

#### 2. Download from GitHub Releases:

- Visit IWE releases
- Download the appropriate binary for your system
- Ensure iwes is in your system PATH

#### 3. Build from Source:

```

git clone https://github.com/iwe-org/iwe.git
cd iwe
cargo build --release --bin iwes
# Copy target/release/iwes to your PATH

```

### Verify Installation

1. Open Neovim in a directory with markdown files
2. Open a .md file
3. Run `:checkhealth iwe` to verify the plugin is working
4. Check `:LspInfo` to see if the IWE LSP server is attached

## Neovim Shortcuts for IWE Actions

### Default Keybindings

The plugin provides these default keybindings (can be customized):

### IWE Feature Keybindings

IWE Feature	Neovim Shortcut	Mode	Description
<b>Code Actions</b>	ca	Normal	Extract/Inline/Transform code actions
<b>Go to Definition</b>	gd	Normal	Go to definition of symbol under cursor
<b>Find References</b>	gr	Normal	Find backlinks to current document
<b>Document Symbols</b>	ds	Normal	Navigate document outline
<b>Workspace Search</b>	ws	Normal	Global search with Telescope
<b>Format Document</b>	f	Normal/Visual	Auto-format document
<b>Rename Symbol</b>	rn	Normal	Rename symbol (including file & references)

### LSP Keybindings

Feature	Shortcut	Description
<b>Hover Info</b>	K	Show information about current element
<b>Signature Help</b>		Show function signature (Insert mode)
<b>Code Action</b>	ca	Show available code actions
<b>Diagnostic Next</b>	]d	Jump to next diagnostic
<b>Diagnostic Previous</b>	[d	Jump to previous diagnostic

### Telescope Integration

Command	Shortcut	Description
:Telescope iwe search	ws	Search through all notes
:Telescope iwe backlinks	wb	Find backlinks to current document
:Telescope iwe links	wl	Browse all links in current document

## Configuration

### Basic Setup

```
require("iwe").setup({
  -- LSP server configuration
  lsp = {
    -- Path to iwes binary (auto-detected if in PATH)
    cmd = { "iwes" },

    -- LSP server settings
    settings = {
      iwe = {
        debug = false,
      },
    },
  },
},
```

```

-- Keybindings (set to false to disable default bindings)
keybindings = {
  enable = true,

  -- Custom keybindings
  code_action = "<leader>ca",
  goto_definition = "gd",
  find_references = "gr",
  document_symbols = "<leader>ds",
  workspace_search = "<leader>ws",
  format_document = "<leader>f",
  rename_symbol = "<leader>rn",
},

-- Telescope integration
telescope = {
  enable = true,

  -- Telescope-specific settings
  search = {
    layout_strategy = "horizontal",
    layout_config = {
      preview_width = 0.6,
    },
  },
},
},
})

```

## Advanced Configuration

```

require("iwe").setup({
  -- LSP configuration
  lsp = {
    cmd = { "iwes" },
    filetypes = { "markdown" },
    root_dir = function(fname)
      return require("lspconfig.util").root_pattern(".iwe")(fname)
      or require("lspconfig.util").find_git_ancestor(fname)
      or vim.loop.os_homedir()
    end,

    -- Custom capabilities
    capabilities = require("cmp_nvim_lsp").default_capabilities(),

    -- LSP server settings
    settings = {
      iwe = {
        debug = vim.env.IWE_DEBUG == "true",
        trace = "off", -- or "messages", "verbose"
      },
    },
  },

  -- Custom handlers
  handlers = {
    ["textDocument/hover"] = vim.lsp.with(vim.lsp.handlers.hover, {
      border = "rounded",
    })
  }
})

```

```

    }),
  },
},

-- Disable default keybindings and set custom ones
keybindings = {
  enable = false, -- Disable defaults
},

-- Telescope customization
telescope = {
  enable = true,
  extensions = {
    iwe = {
      search = {
        prompt_title = "IWE Search",
        results_title = "Documents",
      },
      backlinks = {
        prompt_title = "Backlinks",
        results_title = "References",
      },
    },
  },
},

-- Health check configuration
health = {
  check_iwes_binary = true,
  check_iwe_config = true,
},
})

-- Custom keybindings
local map = vim.keymap.set
map("n", "<leader>ia", "<cmd>lua vim.lsp.buf.code_action()<cr>", { desc = "IWE Code Actions" })
map("n", "<leader>ig", "<cmd>lua vim.lsp.buf.definition()<cr>", { desc = "IWE Go to Definition" })
map("n", "<leader>ir", "<cmd>lua vim.lsp.buf.references()<cr>", { desc = "IWE Find References" })
map("n", "<leader>is", "<cmd>Telescope iwe search<cr>", { desc = "IWE Search" })
map("n", "<leader>ib", "<cmd>Telescope iwe backlinks<cr>", { desc = "IWE Backlinks" })
map("n", "<leader>if", "<cmd>lua vim.lsp.buf.format()<cr>", { desc = "IWE Format" })
map("n", "<leader>in", "<cmd>lua vim.lsp.buf.rename()<cr>", { desc = "IWE Rename" })

```

## Which-Key Integration

If you use which-key.nvim, add descriptions for IWE commands:

```

require("which-key").register({
  ["<leader>i"] = {
    name = "IWE",
    a = "Code Actions",
    g = "Go to Definition",
    r = "Find References",
    s = "Search",
  },
})

```



```
    b = "Backlinks",  
    f = "Format Document",  
    n = "Rename",  
  },  
})
```

## Usage Examples

### Extracting a Section

1. Place cursor on a header line (e.g., `## Section Title`)
2. Press `<leader>ca` to open code actions
3. Select “Extract section” from the list
4. Neovim will create a new buffer with the extracted content

### Following Links

1. Place cursor on any markdown link
2. Press `gd` to follow the link
3. Use `<C-o>` to return to the previous location

### Finding Backlinks

1. In any document, press `gr` or `<leader>wb`
2. Telescope will show all documents linking to the current one
3. Use arrow keys to navigate, Enter to open

### Global Search with Telescope

1. Press `<leader>ws` to open IWE search
2. Start typing to search across all documents
3. Results show document paths and matching content
4. Use `<C-p>` preview to see content without opening

### Text Transform Actions (if configured)

1. Select text in visual mode
2. Press `<leader>ca` to show code actions
3. Choose from available transform actions
4. The text will be processed and replaced

## Telescope Commands

### Available Commands

```
" Search through all notes  
:Telescope iwe search
```

```
" Find backlinks to current document  
:Telescope iwe backlinks
```

```
" Browse links in current document  
:Telescope iwe links
```

```
" Show document symbols/outline  
:Telescope lsp_document_symbols
```

```
" Search workspace symbols  
:Telescope lsp_workspace_symbols
```

Key	Action
	Open selected item
	Open in horizontal split
	Open in vertical split
	Open in new tab
	Scroll preview up
	Scroll preview down
	Send to quickfix list

## Health Check

Run health checks to verify your setup:

```
:checkhealth iwe
```

This will check:

- IWE plugin installation
- iwe binary availability
- LSP server configuration
- Telescope integration
- IWE project configuration

## Troubleshooting

### Common Issues

#### 1. LSP Server Not Starting

```
# Check if iwe is in PATH
which iwe
```

```
# Check LSP server status
:LspInfo
```

```
# View LSP logs
:LspLog
```

#### 2. Telescope Not Working

```
-- Ensure telescope is loaded
require("telescope").load_extension("iwe")
```

#### 3. Keybindings Not Working

- Check if default keybindings are enabled in config
- Verify no conflicts with other plugins
- Use `:verbose map <key>` to check key mappings

#### 4. Performance Issues

- Check `:IweStatus` for server information
- Consider workspace size and complexity
- Enable debug mode temporarily: `IWE_DEBUG=true nvim`

## Debug Mode

Enable debug logging:

```
# Start Neovim with debug mode
IWE_DEBUG=true nvim
```

```
# Or set in Neovim
:lua vim.env.IWE_DEBUG = "true"
:LspRestart
```

Debug logs will be written to `iwe.log` in your working directory.

## Getting Help

- **Plugin Repository:** [iwe.nvim Issues](#)
- **Main Project:** [IWE Issues](#)
- **Discussions:** [Community Support](#)
- **Documentation:** [Full Wiki](#)

## Integration with Other Plugins

### nvim-cmp (Autocompletion)

```
require("cmp").setup({
  sources = {
    { name = "nvim_lsp" }, -- Includes IWE completions
    { name = "buffer" },
    { name = "path" },
  },
})
```

### nvim-treesitter

```
require("nvim-treesitter.configs").setup({
  ensure_installed = { "markdown", "markdown_inline" },
  highlight = { enable = true },
})
```

### gitsigns.nvim

IWE works well with git integration for version control of your knowledge base.

## Best Practices for Neovim

1. **Use Workspace Sessions:** Save and restore IWE workspace sessions
2. **Configure LSP Properly:** Ensure proper root directory detection
3. **Leverage Telescope:** Use fuzzy finding for efficient navigation
4. **Set Up Health Checks:** Regular `:checkhealth iwe` for maintenance
5. **Customize Keybindings:** Adapt shortcuts to your workflow
6. **Use Splits and Tabs:** Work with multiple documents simultaneously
7. **Enable Auto-Save:** Use `:set autowrite` to prevent data loss
8. **Integrate with Git:** Version control your knowledge base
9. **Configure Completion:** Set up `nvim-cmp` for link auto-completion
10. **Use Which-Key:** Document your IWE keybindings for easy reference

## Zed

IWE integrates with Zed editor through an official extension that provides LSP support for markdown files.

## Installation

### From Zed Extensions

1. Open Zed
2. Open the Extensions panel (Cmd+Shift+X on macOS)

3. Search for “IWE”
4. Click Install

The extension will automatically download the `iwes` language server binary when first activated.

### Manual Installation

If you prefer to manage the binary yourself:

1. Install `iwes` using one of the methods from the How to Install
2. Ensure `iwes` is available in your `$PATH`
3. Install the IWE extension from Zed Extensions

The extension will use the system `iwes` binary if available, otherwise it downloads from GitHub releases.

### Setup

#### Enable for Specific Projects

To enable IWE only for your notes directory (not all markdown files), create a `.zed/settings.json` in your notes root:

```
{
  "lsp": {
    "iwe": {
      "binary": {
        "path": "iwes"
      }
    }
  }
}
```

#### Initialize IWE

Create an IWE project in your notes directory:

```
cd ~/notes
iwes init
```

This creates the `.iwe/config.toml` configuration file.

### Usage

Please refer to the How to Use with Your Text Editor for a quick reference.

### Common Keybindings

Action	Keybinding
Go to definition (follow link)	F12 or Cmd+Click
Find references (backlinks)	Shift+F12
Code actions	Cmd+.
Format document	Cmd+Shift+I
Document symbols (outline)	Cmd+Shift+O
Workspace symbols (search)	Cmd+T
Rename	F2

### Code Actions

To use IWE code actions (extract, inline, attach, etc.):

1. Place cursor on the target content
2. Press Cmd+. to open code actions menu
3. Select the desired action

## Troubleshooting

### Extension Not Working

1. Check that the extension is installed and enabled
2. Verify that your notes directory has `.iwe/config.toml`
3. Check Zed's LSP logs for errors

### Binary Not Found

If the extension fails to download the binary:

1. Install iwes manually from How to Install
2. Ensure it's in your \$PATH
3. Restart Zed

## Platform Support

The extension supports:


- macOS (Apple Silicon and Intel)
- Linux (x86\_64 and aarch64)

Windows support is planned for a future release.

## Examples

### Basic Journal Example

Lets take this Markdown journal as an example.

 journal-2025.md

# Journal, 2025

## Week 3 - Coffee week

This week, I tried three types of coffee: the **cappuccino** with its bold espresso and frothy milk offering a delightful texture, the **latte** which envelops espresso and milk in a comforting embrace perfect for leisurely mornings, and the **cortado**, a balanced blend of espresso and milk that brings peace to the taste buds.

### Jan 26, 2025 - Cappuccino

It's cappuccino day. The classic Italian masterpiece, where espresso meets a frothy cloud of milk, creating a delightful contrast of bold and creamy. It's like sipping on a caffeine-infused cloud, perfect for anyone wanting to add a little texture to their daily routine.

### Jan 25, 2025 - Latte


As warm as a hug from an old friend, the latte wraps espresso and milk in a snug embrace. With a canvas for barista art, it's not just a drink, but a little piece of serenity in a cup for those more leisurely mornings when taking it slow is the only option.

### Jan 24, 2025 - Cortado

I had an amazing cortado today. It's when espresso and milk meet halfway in a charming truce, the cortado emerges. It's the perfect compromise, bringing balance to your coffee routine and peace to your taste buds.

This kind of a document can grow very fast. IWE can transform it by *collapsing* sections into *Inclusion Links*. This transformation maintains the document hierarchy while reducing level of details.

**Inclusion links** are standalone markdown links that create parent-child relationships in your knowledge graph. When a link like [Topic](topic) appears on its own line, the linked document becomes a *child* of the current document. A document can have multiple parents (polyhierarchy), enabling the same note to appear in different contexts.

 journal-2025.md

# Journal, 2025

## Week 3 - Coffee week


This week, I tried three types of coffee: the **cappuccino** with its bold espresso and frothy milk offering a delightful texture, the **latte** which envelops espresso and milk in a comforting embrace perfect for leisurely mornings, and the **cortado**, a balanced blend of espresso and milk that brings peace to the taste buds.


[Jan 26, 2025 - Cappuccino](jan-26)


[Jan 25, 2025 - Latte](jan-25)

[Jan 24, 2025 - Cortado](jan-24)


And three daily files:

 jan-26.md

 jan-25.md


 jan-24.md


You can repeat this again, adding as many levels as necessary


 journal-2025.md


# Journal, 2025

[Week 3 - Coffee week](2025-W3)

 2025-W3.md

 jan-26.md

 jan-25.md

 jan-24.md

As a result of this decomposition, each document is much simpler while the original hierarchy is preserved. It's also a perfectly valid markdown with no additional syntax.

IWE supports automated actions for graph transformations like this and it can just as easily reconstruct the **original** document by combining the extracted content together preserving correct headings structure.

## About the project

### Why It Exists

I've always been a big fan of modern text editors like Neovim and Zed, and I've longed to manage my Markdown notes in a way similar to how I write code. I wanted features like "Go To Definition" for diving into details, "Extract note" refactoring for breaking down complex documents into smaller more manageable notes, and autocomplete notes linking.

All modern editors support the Language Server Protocol (LSP), which enhances text editors with IDE-like capabilities. This was exactly what I wanted for my Markdown notes.

So, I developed an LSP called IWE. It includes essential features such as note search, link navigation, autocomplete, backlink search, and some unique capabilities like:

- Creating a nested notes hierarchy.
- Extract/inline refactoring for improved note management.
- Code actions for various text transformations.
- And more

IWE allows you to build a notes library that can support basic journaling as well as GTD, Zettelkasten, PARA, you name it methods of note-taking. IWE does not enforce any structure on your notes library. It doesn't care about your file names preference. It's only give you tools to manage the documents and connections between them with least possible effort automating routine operations such as formatting, keeping link titles up to date and many other.

This is all possible because of IWE's unique Architecture. IWE loads notes into an in-memory graph structure, which understands the hierarchy of headers and lists. This allows it to go through the graph, reorganize, and transform the content as needed using graph iterators.

### Unique Features

IWE combines powerful knowledge management with developer-focused tooling, offering unique capabilities not found in other PKM solutions:

#### Graph-based Transformations

- **Extract/embed notes operations:** Use LSP code actions to extract sections into separate notes or inline referenced content
- **Section-to-list and list-to-section conversions:** Transform document structure with a single click
- **Sub-sections extraction:** Break complex notes into manageable, linked components
- **Reference inlining:** Convert linked content to quotes or embed sections directly

#### External Command Integration

- **Configurable command pipeline:** Connect to any CLI tool with custom templates
- **Block-level transformations:** Apply transformations to specific sections with full context awareness
- **Template-based input:** Customize command behavior for different content types
- **Context-aware processing:** Commands receive document structure and relationships

#### Developer-Focused Architecture

- **Rust-powered performance:** Built with Rust for speed and reliability, handling thousands of files instantly
- **Shared core library:** CLI and LSP server share the same robust domain model
- **Rich graph processing:** Advanced algorithms for document relationships and transformations

- **Cross-platform:** Works identically across all supported operating systems

### Advanced Markdown Normalization

- **Batch operations:** Normalize thousands of files in under a second
- **Auto-formatting on save:** Fix link titles, header levels, list numbering automatically
- **Header hierarchy management:** Maintain consistent document structure
- **Link title synchronization:** Keep link text in sync with target document titles

### Hierarchical Note Support

- **Context-aware search:** Find notes by understanding their position in the knowledge graph
- **Inlay hints:** See parent note context without leaving your current document
- **Flexible file organization:** Supports both flat Zettelkasten and hierarchical structures
- **Path-based navigation:** Multiple ways to reach the same content through different conceptual paths

### Cross-Editor LSP Integration

- **Native LSP support:** Works with VSCode, Neovim, Zed, Helix, and any LSP-compatible editor
- **Consistent experience:** Same features and performance across all editors
- **No vendor lock-in:** Switch editors without losing functionality

IWE also includes a comprehensive CLI utility for batch operations, document generation, and graph visualization.

The core differentiator is the shared library architecture between CLI and LSP components. This rich domain model enables easy construction of new graph transformations and ensures consistency across all interfaces. You can learn more in the Data Model documentation.

### Detailed Comparisons

#### IWE vs markdown-oxide

**markdown-oxide** is a solid PKM LSP server focused on basic knowledge management:

Feature	IWE	markdown-oxide
<b>Graph Transformations</b>	✅ Extract/embed sections, convert lists↔sections, inline references	❌ Basic linking only
<b>External Commands</b>	✅ Configurable CLI tools (supports AI agents, scripts, Unix tools)	❌ No external command features
<b>Performance</b>	✅ Rust-based, handles thousands of files instantly	✅ Good performance
<b>Batch Operations</b>	✅ CLI for bulk normalization and transformations	❌ LSP-only approach
<b>Editor Support</b>	✅ VSCode, Neovim, Zed, Helix	✅ VSCode, Neovim, Helix, Zed
<b>Auto-formatting</b>	✅ Comprehensive normalization on save	✅ Basic formatting
<b>Daily Notes</b>	✅ Supported using “attach” code action	✅ Dedicated daily notes support



Feature	IWE	markdown-oxide
<b>Backlinks</b>	✅ Via graph processing	✅ Native backlink support

**IWE's advantage:** Advanced graph operations, external command integration, and comprehensive CLI tooling make it superior for complex knowledge work and developer workflows.

### IWE vs Obsidian

**Obsidian** is a popular GUI-based PKM tool with strong visualization capabilities and an extensive plugin ecosystem:

Feature	IWE	Obsidian
<b>Editor Integration</b>	✅ Works with your preferred text editor (VSCode, Neovim, Zed, Helix)	❌ Proprietary editor only
<b>Cost</b>	✅ Completely free and open source	⚠️ Free for personal use, \$8/month for sync, \$16/month for publishing
<b>Performance</b>	✅ Rust-powered, instant operations on thousands of files	⚠️ Electron-based, can be slower with large vaults
<b>Graph Transformations</b>	✅ Automated extract/embed operations, section-to-list conversions	❌ Manual linking and organization
<b>External Commands</b>	✅ Configurable CLI tools (supports AI agents, scripts, Unix tools)	⚠️ Limited, requires third-party plugins (Text Generator, Smart Connections)
<b>Inclusion Links</b>	✅ Native support with automatic linking	⚠️ Available via plugins
<b>Auto-formatting</b>	✅ Comprehensive markdown normalization on save	⚠️ Basic formatting, requires plugins for advanced normalization
<b>Batch Operations</b>	✅ CLI for bulk transformations and normalization	❌ No batch operation capabilities
<b>Cross-platform</b>	✅ Consistent across all platforms	✅ Good cross-platform support
<b>Graph Visualization</b>	⚠️ CLI-based dot export (can generate visual graphs)	✅ Interactive graph view with customizable styling
<b>Plugin Ecosystem</b>	⚠️ Limited to LSP capabilities and CLI extensions	✅ Rich plugin marketplace with 1000+ community plugins
<b>Learning Curve</b>	⚠️ Requires basic terminal knowledge and editor setup	✅ GUI-friendly with intuitive interface
<b>Sync &amp; Collaboration</b>	✅ Git-based sync (free), works with any Git hosting	⚠️ Obsidian Sync (\$8/month) or manual Git setup

Feature	IWE	Obsidian
<b>Publishing</b>	✅ Export to various formats via CLI	⚠️ Obsidian Publish (\$16/month) or manual export
<b>Mobile Support</b>	❌ Desktop/terminal only	✅ Native mobile apps with sync

#### ##### When to Choose IWE

- **Developer workflows:** You want to stay in your preferred code editor
- **Large repositories:** You need instant performance with thousands of markdown files
- **Automation-heavy workflows:** You want CLI-powered transformations and batch operations
- **Cost-conscious:** You want a completely free, open-source solution
- **Technical users:** You're comfortable with CLI tools and LSP setup
- **Git-based workflows:** You prefer version control over proprietary sync

#### ##### When to Choose Obsidian

- **GUI preference:** You prefer visual interfaces over terminal-based tools
- **Plugin ecosystem:** You want access to hundreds of community plugins
- **Mobile access:** You need to access notes on phones/tablets
- **Interactive visualization:** You want to explore your knowledge graph visually
- **Non-technical users:** You want a user-friendly setup without terminal configuration
- **Rich formatting:** You need advanced formatting, canvas features, or embedded media

**Key Philosophical Difference:** IWE is editor-agnostic and developer-focused, designed to integrate with existing technical workflows. Obsidian is a comprehensive PKM environment with its own ecosystem, better suited for users who want an all-in-one solution with visual interfaces and extensive customization through plugins.

#### IWE vs zk.nvim/telekasten.nvim

**zk.nvim** and **telekasten.nvim** are Neovim-specific Zettelkasten solutions:

Feature	IWE	zk.nvim/telekasten
<b>Editor Support</b>	✅ VSCode, Neovim, Zed, Helix, others	❌ Neovim only
<b>Graph Transformations</b>	✅ Automated extract/embed, structural changes	❌ Basic note creation and linking
<b>External Commands</b>	✅ Configurable CLI tools (supports AI agents, scripts, Unix tools)	❌ Manual workflows only
<b>Performance</b>	✅ Rust-powered LSP	⚠️ Lua-based, editor-dependent
<b>Batch Operations</b>	✅ CLI for bulk operations	❌ One-note-at-a-time workflow
<b>Auto-formatting</b>	✅ Built-in normalization	❌ Requires external tools
<b>Note Templates</b>	✅ Note templates supported via "Attach" command	✅ Static templates
<b>Search Integration</b>	✅ LSP-based with any picker	✅ Telescope/fzf integration

Feature	IWE	zk.nvim/telekasten
<b>Installation</b>	✅ Single binary + editor extension	⚠️ Complex Neovim plugin setup

**IWE’s advantage:** Works across all editors, provides powerful automation, and offers external command integration. zk.nvim/telekasten are better for Neovim purists who prefer simple, manual workflows.

**Why Choose IWE?**

IWE is the **only tool** that combines:

- 🚀 **Performance:** Rust-powered speed that handles thousands of files instantly
- 🧰 **Extensibility:** External command integration with contextual templates for enhanced workflows
- 🔧 **Flexibility:** Works with VSCode, Neovim, Zed, Helix, and any LSP-compatible editor
- ⚡ **Power:** Advanced graph transformations and batch operations
- 🛠️ **Developer Focus:** CLI + LSP architecture designed for technical workflows

IWE is powerful enough for complex knowledge work, fast enough for large repositories, and flexible enough to adapt to any workflow or editor preference. Whether you’re a researcher managing thousands of notes, a developer documenting complex systems, or a writer organizing interconnected ideas.

**Design Principles**

IWE is text graph management assistant. Any text graph. The graph structure is not imposed.

- Do not assume any particular graph structure or file naming convention.  
It is up to the user to decide on these details.
- The goal is to minimize routine operations.  
Keeping the graph consistent should require the least possible amount of effort. The text formatting needs to be automated.
- Focus on building blocks, not specific features.  
All simple operations as a “daily note” can be implemented at the editor level. IWE is merely a tool for navigating and changing text graphs.

**Architecture**

IWE’s data model is built around a **graph-based representation** of markdown documents, where each structural element becomes a node in an interconnected graph. This design enables sophisticated document operations, cross-references, and transformations that go far beyond traditional markdown processing.

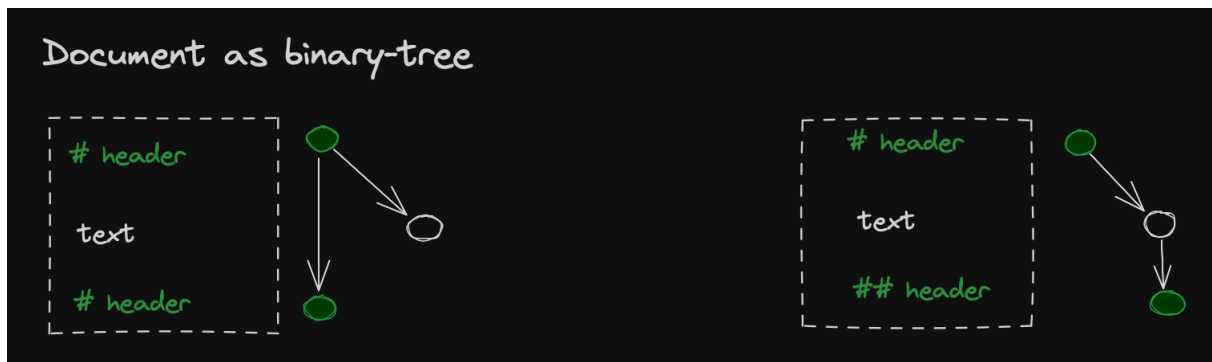
**Data Model**

**Graph-Based Document Representation**

Unlike traditional parsers that work with document trees, IWE represents text as a **directed graph** where every **header**, **paragraph**, **list**, **list item**, **code block**, **table**, and **reference** becomes a **node**. Each node can have up to two primary relationships:

- **next-element:** Points to the sibling node at the same hierarchical level
- **child-element:** Points to the first child node (for container elements)

This creates a hybrid tree-graph structure that preserves both document hierarchy and enables complex cross-document relationships.



## Arena-Based Memory Management

IWE uses an **arena pattern** for efficient memory management and fast graph operations:

```
#[derive(Clone, Default)]
pub struct Arena {
    nodes: Vec<GraphNode>, // All graph nodes stored contiguously
    lines: Vec<Line>,       // Text content stored separately
}
```

### Benefits of arena storage:

- **Fast access:**  $O(1)$  node lookup using `NodeId` as array index
- **Memory efficiency:** Contiguous storage reduces memory fragmentation
- **Cache locality:** Related nodes stored close together in memory
- **Safe deletion:** Empty nodes marked rather than removed

## Node Types and Structure

### GraphNode Enumeration

IWE defines 9 distinct node types, each optimized for specific markdown elements:

```
pub enum GraphNode {
    Empty, // Deleted/placeholder nodes
    Document(Document), // Root document container
    Section(Section), // Headers (h1-h6)
    Quote(Quote), // Blockquotes
    BulletList(BulletList), // Unordered lists
    OrderedList(OrderedList), // Numbered lists
    Leaf(Leaf), // Paragraphs and simple blocks
    Raw(RawLeaf), // Code blocks and raw content
    HorizontalRule(HorizontalRule), // Horizontal rules
    Reference(Reference), // Inclusion links to other documents
    Table(Table), // Markdown tables
}
```

### Node Relationships and Navigation

Each node (except Document and Empty) contains:

- **id:** Unique identifier within the graph
- **prev:** Reference to previous sibling or parent
- **next:** Optional reference to next sibling
- **child:** Optional reference to first child (container nodes only)

## Navigation patterns:

- **Siblings:** Follow next pointers horizontally
- **Children:** Follow child pointer then next for all children
- **Parent:** Use prev pointer and traverse up

## Content Storage Separation

Text content is stored separately from structure in Line objects:

```
pub struct Line {  
    id: LineId,  
    inlines: GraphInlines, // Vector of inline elements (text, links, formatting)  
}
```

This separation enables:

- **Structure reuse:** Multiple nodes can reference same content
- **Efficient updates:** Content changes don't affect structure
- **Memory optimization:** Structure and content cached independently

## Document Processing Pipeline

### 1. Markdown Parsing (DocumentBlock Creation)

Raw markdown is first parsed into intermediate DocumentBlock representations:

```
pub enum DocumentBlock {  
    Plain(Plain),           // Plain text paragraphs  
    Para(Para),             // Regular paragraphs  
    CodeBlock(CodeBlock),   // Fenced code blocks  
    Header(Header),         // Headers with level and content  
    BulletList(BulletList), // List containers  
    Table(Table),           // Table structures  
    // ... additional block types  
}
```

### 2. Graph Construction (DocumentBlock → GraphNode)

The SectionsBuilder transforms DocumentBlock elements into graph nodes:

```
// High-level transformation process  
DocumentBlock::Header(header) → GraphNode::Section(section)  
DocumentBlock::Para(para) → GraphNode::Leaf(leaf)  
DocumentBlock::BulletList(list) → GraphNode::BulletList(bulletlist)
```

## Key transformations:

- **Headers become Sections:** With child relationships to content
- **Lists become containers:** With children for each list item
- **Paragraphs become Leaves:** Terminal nodes with text content
- **Code blocks become Raw nodes:** With language and content metadata

### 3. Reference Resolution and Indexing

After graph construction, the RefIndex system processes all references:

```
pub struct RefIndex {  
    block_references: HashMap<Key, HashSet<NodeId>>, // [[note]] references  
    inline_references: HashMap<Key, HashSet<NodeId>>, // [[link]](note) references  
}
```

## Key System and Cross-References

### Document Identification

Each document is identified by a Key - a path-based identifier:

```
pub struct Key {  
    pub relative_path: Arc<String>, // e.g., "folder/document"  
}
```

#### Key features:

- **Path-based:** Hierarchical organization support
- **Reference counting:** Arc enables efficient cloning
- **Extension handling:** Automatic .md extension management
- **Relative linking:** Support for ../parent/document syntax

### Reference Types

IWE supports three reference types:

1. **Regular markdown links:** [text](document.md)
2. **Wiki-style links:** [[document]]
3. **Piped wiki links:** [[document|display text]]

Each reference type is preserved and can be normalized or converted as needed.

## Graph Operations and Algorithms

### Tree Collection

Converting graph sections to tree structures for processing:

```
// Collect a complete tree starting from a node  
let tree = graph_node_pointer.collect_tree();  
  
// Tree provides hierarchical access to content  
for child in tree.children() {  
    process_content(child);  
}
```

### Squashing (Content Extraction)

Extract content at limited depth with proper hierarchy flattening:

```
// Extract content up to depth 2  
let squashed = graph.squash(&document_key, 2);  
  
// Headers are flattened: h1 → h2, h2 → h3, etc.  
// Content preserved with adjusted hierarchy
```

### Path Generation

Generate navigable paths through the document graph:

```
pub struct NodePath {  
    ids: Vec<NodeId>, // Sequence of nodes forming path  
    target: NodeId,   // Final destination node  
}  
  
// Paths enable:  
// - Search result ranking
```

```
// - Navigation breadcrumbs
// - Content organization
```

## Data Flow Architecture

### CLI Operations

CLI commands operate directly on the graph:

```
// Normalization: Rewrite all documents with consistent formatting
fn normalize() { graph.export() → filesystem }

// Export: Generate visualization formats (DOT, etc.)
fn export() { graph → GraphData → DOTExporter }

// Tree: Display document hierarchy
fn tree() { graph.paths() → tree structure → markdown/keys/json }

// Squash: Extract partial content at specified depth
fn squash() { graph.squash(key, depth) → markdown }
```

### LSP Server Integration

The LSP server maintains a live Database wrapper around the graph:

```
pub struct Database {
    graph: Graph,                // Core graph structure
    content: HashMap<Key, Content>, // Original markdown content
    paths: Vec<SearchPath>,      // Pre-computed search paths
}
```

### Real-time operations:

- **Document updates:** Incremental graph rebuilding
- **Reference resolution:** Live link validation
- **Search:** Fuzzy matching against pre-computed paths
- **Completion:** Context-aware suggestions based on graph structure

### Memory and Performance Characteristics

#### Graph construction:

- **Parallel processing:** Rayon integration for multi-document parsing
- **Incremental updates:** Only affected nodes rebuilt on changes
- **Memory efficiency:** Arena pattern minimizes allocation overhead

#### Search performance:

- **Pre-computed paths:** Search index built once, queried repeatedly
- **Fuzzy matching:** SkimMatcher for intelligent search ranking
- **Parallel search:** Multi-threaded query processing

### Indexing and Reference Systems

#### Reference Index Structure

The RefIndex maintains bidirectional reference mappings:

```
impl RefIndex {
    // Find all nodes that reference a specific document
    pub fn get_block_references_to(&self, key: &Key) -> Vec<NodeId>
```

```

// Find all inline references (links) to a document
pub fn get_inline_references_to(&self, key: &Key) -> Vec<NodeId>

// Recursively index a node and all its children
pub fn index_node(&mut self, graph: &Graph, node_id: NodeId)
}

```

### Indexing process:

1. **Graph traversal:** Depth-first traversal of all nodes
2. **Reference extraction:** Parse inline content for links
3. **Bidirectional mapping:** Build forward and reverse reference maps
4. **Incremental updates:** Re-index only changed portions

### Search Path Generation

Search paths provide hierarchical navigation:

```

pub struct SearchPath {
    pub search_text: String,    // Concatenated plain text for matching
    pub node_rank: usize,      // Importance ranking
    pub key: Key,               // Source document
    pub root: bool,            // Is document root
    pub line: u32,              // Line number in source
    pub path: NodePath,         // Complete navigation path
}

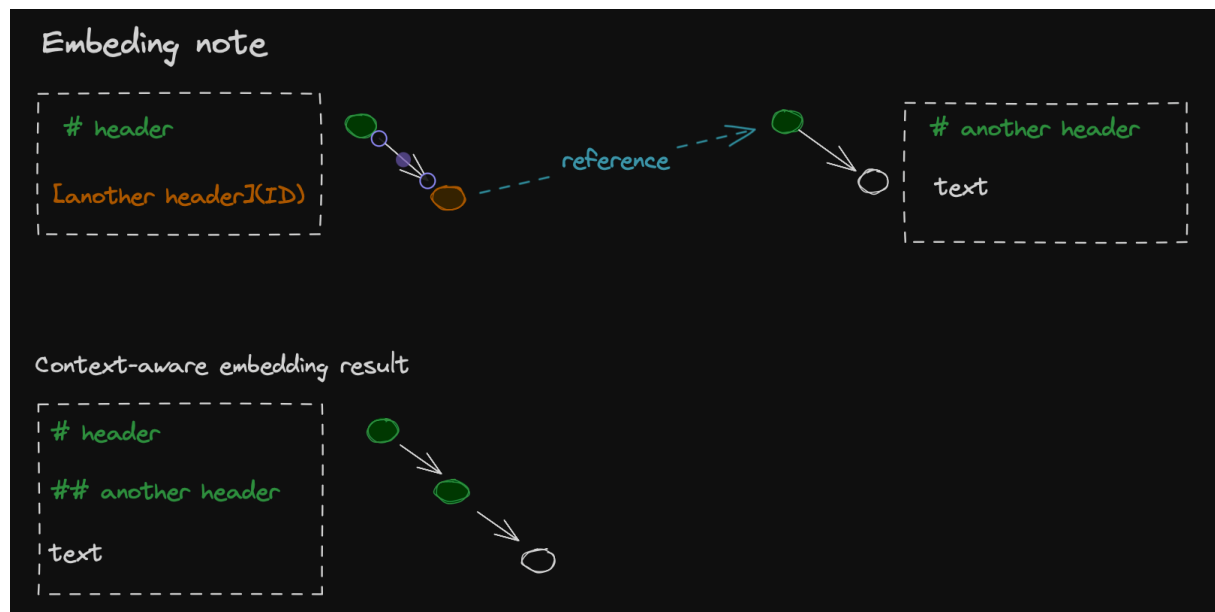
```

### Ranking algorithm:

- **Content depth:** Deeper content ranked lower
- **Reference count:** More referenced content ranked higher
- **Document position:** Earlier content ranked higher
- **Search relevance:** Fuzzy match score integration

### Advanced Features

#### Notes Embedding



IWE can embed referenced documents while preserving structure:



Embedding process:

1. Identify reference nodes
2. Load target document graph
3. Adjust header levels for context
4. Insert content maintaining hierarchy

#### Header level adjustment:

- Embedded under h2 → all headers +2 levels
- h1 becomes h3, h2 becomes h4, etc.
- Maintains document structure integrity

#### Graph Transformations

All document operations are implemented as graph transformations:

- **Normalization:** Graph → normalized graph → markdown
- **Reference resolution:** Update reference targets across graph
- **Content extraction:** Subgraph extraction with proper boundaries
- **Document merging:** Graph composition with conflict resolution

#### Parallel Processing

IWE leverages Rayon for parallel operations:

```
// Parallel document processing
let documents: Vec<(Key, Document)> = content
    .par_iter()
    .map(|(k, v)| (Key::name(k), reader.document(v)))
    .collect();

// Parallel search path generation
let search_paths = self.paths()
    .par_iter()
    .map(|path| generate_search_path(path))
    .collect();
```

This architecture enables IWE to handle large document collections efficiently while maintaining real-time responsiveness for editor integration.