IWE

Installation

How to install

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

From Crates.IO

- Rust and Cargo must be installed on your system. You can get them from rustup.rs.

IWE is available at 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

- 🤖 **AI-Powered Text Generation**: Generate, rewrite, or modify text using configurable AI 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
- 📥 **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
AI Text Generation	Code Action	Generate, rewrite, or modify text using AI
Text Transformation	Code Action	Convert lists to headers, change list types
Link Navigation	Go To Definition	Follow markdown links to target documents
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:

```markdown
# 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:

```markdown
# 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:

```markdown
# 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

AI-Powered Actions

IWE supports configurable AI commands that can:

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

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

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

[actions.rewrite]
title = "Improve Text"
model = "default"
context = "Document"
prompt_template = "Improve this text: {{context}}"
```

Configuration

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 AI**: Set up AI actions that match your writing workflow
7. **Use Inlay Hints**: Enable hints to understand document relationships at a glance

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. **Normalize all documents**: `iwe normalize`
3. **View document paths**: `iwe paths`
4. **Export graph visualization**: `iwe export dot`

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 (0-3, default: 0)
  - `0`: Minimal output
  - `1`: Basic progress information
  - `2`: Detailed operation logs
  - `3`: Debug-level information
- `-h`, `--help`: Show help information

Commands Reference

`iwe init`

Initializes the current directory as an IWE project.

```
iwe init
```

What it does:

- Creates `.iwe/` marker directory
- Generates default `config.toml` configuration
- Sets up the project structure for IWE operations

Example:

```
cd ~/my-notes
iwe init
# Creates .iwe/config.toml with default settings
```

`iwe normalize`

Performs comprehensive document normalization across all markdown files.

```
iwe normalize [OPTIONS]
```

Operations performed:

- Updates link titles to match target document headers
- Adjusts header levels for consistent hierarchy
- Renumbers ordered lists
- Fixes markdown formatting (newlines, indentation)
- Standardizes list formatting
- Normalizes document structure

Options:

- `-v, --verbose <LEVEL>`: Increase output detail

Example:

```
# Basic normalization
iwe normalize

# With detailed output
iwe normalize -v 2
```

⚠️ **Important:** Always backup your files before running normalization, especially the first time.

`iwe paths`

Displays all possible navigation paths in your document graph.

```
iwe paths [OPTIONS]
```

Options:

- `-d, --depth <DEPTH>`: Maximum path depth to explore (default: 4)
- `-v, --verbose <LEVEL>`: Verbosity level

Output format: Shows hierarchical paths through your documents, revealing connection patterns and document relationships.

Example:

```
# Show paths up to depth 4
iwe paths

# Show deeper paths
iwe paths --depth 6

# With verbose output
iwe paths -v 1 --depth 3
```

`iwe contents`

Lists root documents (notes without parent references).

```
iwe contents [OPTIONS]
```

Purpose: Identifies entry points in your knowledge graph - documents that aren't referenced by others, potentially serving as main topics or starting points.

Options:

- `-v, --verbose <LEVEL>`: Verbosity level

Example:

```
iwe contents
```

`iwe squash`

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

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

Required:

- `-k, --key <KEY>`: Starting document key/identifier to squash from

Options:

- `-d, --depth <DEPTH>`: How deep to traverse links (default: 2)
- `-v, --verbose <LEVEL>`: Verbosity level

What it does:

- Starts from the specified document
- Traverses linked documents up to specified depth
- Combines content into a single markdown document
- Converts block references to inline sections
- Maintains document structure and hierarchy

Examples:

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

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

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

Example PDF generated using `squash` command and typst

`iwe export`

Exports graph structure in various formats for visualization and analysis.

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

Available formats:

- `dot`: Graphviz DOT format for graph visualization

Options:

- `-k, --key <KEY>`: Filter to specific document and its connections (default: exports all root notes)
- `-d, --depth <DEPTH>`: Maximum depth to include (default: 0 = unlimited)
- `--include-headers`: Include section headers and create detailed subgraphs
- `-v, --verbose <LEVEL>`: Verbosity level

DOT Export 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 and headers
iwe export dot --key "research" --depth 3 --include-headers
```

Using DOT output:

```
# 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

# Interactive visualization
iwe export dot | dot -Tsvg | firefox /dev/stdin
```

Workflow Examples

Daily Maintenance

```
# Update all document formatting and links
iwe normalize

# Check document structure
iwe paths --depth 5
```

Content Analysis

```
# Find entry points
iwe contents

# Visualize specific topic area
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 --key "research-index" --depth 4 > consolidated-research.md

# Generate presentation material
iwe squash --key "project-summary" --depth 2 > project-overview.md
```

Large Library Management

```
# Process with progress information
iwe normalize -v 1

# Analyze complex relationships
iwe paths --depth 8 -v 1
```

```
# Export detailed visualization
iwe export dot --include-headers --depth 5 > full-graph.dot
```

Configuration

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

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

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

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 Verbosity**: Add `-v 1` or `-v 2` to understand what operations are being performed
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 `contents` to track entry points as your library grows

Troubleshooting

- **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

Configuration

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

Basic Configuration

```
prompt_key_prefix = "prompt"

[markdown]
refs_extension = ""

[library]
path = ""
```

- `prompt_key_prefix`: Prefix for AI prompt keys (default: “prompt”)
- `markdown.refs_extension`: File extension for markdown references (default: empty, uses `.md`)
- `library.path`: Subdirectory for markdown files relative to project root (default: empty, uses root)

AI Models

Define LLM models for AI-powered actions:

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

[models.fast]
api_key_env = "OPENAI_API_KEY"
```

```
base_url = "https://api.openai.com"
name = "gpt-4o-mini"
```

Each model requires:

- `api_key_env`: Environment variable containing API key
- `base_url`: API endpoint URL
- `name`: Model name

Optional parameters:

- `max_tokens`: Maximum tokens for input
- `max_completion_tokens`: Maximum tokens for completion
- `temperature`: Sampling temperature (0.0-1.0)

AI Actions

Define custom AI-powered text editing actions:

```
[actions.rewrite]
title = "Rewrite"
model = "default"
context = "Document"
prompt_template = """
Here's a text that I'm going to ask you to edit...
"""
```

Each action requires:

- `title`: Display name in editor
- `model`: Reference to model name
- `context`: Context type ("Document")
- `prompt_template`: Prompt with `{{context}}`, `{{context_start}}`, `{{context_end}}`, `{{update_start}}`, `{{update_end}}` placeholders

Debug Mode

IWE includes a debug mode, which can be enabled by setting 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. Including logs with your issue report will help us to resolve it faster.

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

Maps of Content

Personal Knowledge Management (PKM) systems revolve around managing a graph of notes. However, every Markdown file is a graph in itself. Let me explain with an example:

```
# Header 1

Paragraph 1
```

Here, `Header 1` is the logical parent of `Paragraph 1`.

```
# Header 1

## Header 2

Paragraph 1
```

In this example, `Paragraph 1` belongs to `Header 2`, which in turn belongs to `Header 1`.

You get the idea: it effectively forms a tree (which is also a graph) of text blocks.

So, why does this matter? Suppose I want to find something in my notes graph. I can achieve better results using context-aware search. For example:

```
# Projects

## My new shiny thing

Paragraph 1
```

If I type "Proj" in the search bar, I should get two matches:

```
Projects > My new shiny thing
Projects
```

And if I type "shiny," the search result should be:

```
Projects > My new shiny thing
```

This way, I gain a bit of context.

Okay, it sounds promising, but how can I scale this to thousands of notes and multiple contexts?

It's simple. Just use the "Maps of Content" (MOC) approach:

```
# Projects

[[My new shiny thing]]

[[The old thing]]

[[The old thing 2]]
```

This will yield similar search results:

```
Projects > My new shiny thing
Projects > The old thing 2
Projects > The old thing
Projects
```

With this approach, you can delve as deep as you like:

```
# Personal

[[Projects]]
```

```
Personal > Projects > My new shiny thing
Personal > Projects > The old thing 2
Personal > Projects > The old thing
Personal > Projects
```

Structure Notes: Beyond Simple Lists

While Maps of Content provide hierarchical organization, you can create even more sophisticated structures using **Structure Notes** - meta-notes that explicitly document relationships between other notes.

Hub Notes as Entry Points

Not every relevant note needs to be listed directly in your main MOCs. Instead, create central hub notes that serve as entry points to specific topics. These hub notes contain links to the most important notes on a subject, which then connect to related materials.

For example, instead of listing every single project-related note in your main Projects MOC, you might have:

```
# Projects

[[Active Projects]] - Current work and ongoing initiatives

[[Project Templates]] - Standardized approaches and methodologies

[[Project Archive]] - Completed projects and lessons learned
```

Each hub note then contains its own detailed connections and references.

Types of Structural Relationships

Structure Notes can capture different types of relationships beyond simple hierarchies:

Sequential Structures

Some knowledge follows logical sequences or chains of reasoning. A Structure Note can map these step-by-step progressions:

```
# Feature Development Process

[[Requirements Gathering]] → [[Design Phase]] → [[Implementation]] → [[Testing]] →
[[Deployment]]

Each phase builds on the previous, with dependencies clearly mapped.
```

Overlapping Connections

Unlike strict trees, knowledge often has cross-connections. A note about “API Design Patterns” might belong both in your “Software Architecture” MOC and your “Web Development” MOC, creating a network structure rather than a simple hierarchy.

Thematic Clustering

Group related concepts that share common themes or applications:

```
# Mental Models for Problem Solving

## Analysis Models

[[Root Cause Analysis]]

[[Five Whys Technique]]

[[Fishbone Diagrams]]

## Systems Models

[[Feedback Loops]]

[[Leverage Points]]

[[Network Effects]]
```

Creating Effective Structure Notes

When building Structure Notes:

1. **Focus on Relationships**: Don't just list notes - explain how they connect and why they belong together
2. **Use Multiple Structures**: Combine hierarchical, sequential, and network structures as appropriate
3. **Maintain Entry Points**: Ensure your most important Structure Notes are linked from main MOCs
4. **Update Regularly**: As you create new notes, update relevant Structure Notes to include them

This approach transforms your knowledge base from a simple collection of linked notes into a sophisticated web of interconnected ideas, making it easier to navigate, discover connections, and generate new insights.

Learn more

- A great explanation of what the structure notes are and how to use them is available here
- MOC's overview is available here

Features

Notes search

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

Search is can be used via 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!

Extract/Inline Notes

The extract note action enables the creation of a new document from a section (header). This involves:

1. Creating a new file containing the selected content.
2. Adding a link to the newly created file.

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

1. Embed the content into the document with Block-reference.
2. Remove the link and injected file.

Both operations automatically adjust the header levels as needed to maintain proper document structure.

Navigation

IWE supports multiple ways to navigate your documents, including:

- **Links Navigation**: Implemented as Go To Definition LSP command. **Table of Contents**: Provided as Document Symbols to the editor.

- **Backlinks List**: A backlinks list compiles references or citations linking back to the current document.

Search

Search is one of the key features. IWE creates all possible document paths by considering the block-references structure. This means it can come up with lists like:

```
Readme - Features
Readme - Features - Navigation
Readme - Features - Search
```

And provide this list to your text editor as Workspace Symbols.

This allows for context-aware fuzzy searching, making it easier for you to find what you need.

The search results are ordered by page-rank which is based on the number of references to the target note.

Text structure normalization / formatting

LSP offers **auto-formatting**, which typically kicks in when you save your work. This feature helps tidy things up. Here's what gets cleaned up:

1. Updating link titles to the header of the linked document.
2. Adjusting header levels to ensure tree structure.
3. Updating the numbering of the ordered lists.
4. Fixing newlines, indentations in lists, and much more.

Inlay hints

Note header inlay hint shows parent document title and links counter.

Block reference hint inlay hint includes list of direct parent notes adding essential context of the note link.

Auto-complete

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

Text manipulation

IWE offers a range of actions to help you perform context-aware transformations on your notes. The actions can be called with the "code actions" LSP menu of your editor. Some of the actions available are:

- Transforming lists to headers/sections and vice-versa.
- Changing list type (bullet/ordered).

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:

```markdown
## First Header

# Second Header
```

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

```markdown
# First Header
# Second Header
```

Removing unnecessary levels

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

```markdown
# First header

### Second header
```

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

```markdown
# First header

## Second header
```

Files renaming

With IWE, you can rename the note file and automatically update all the references throughout your entire library using the `rename` LSP refactoring feature.

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

```bash
# 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:

```bash
# 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`

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 you 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 cheat sheet for a quick reference.

> TODO: add specific examples and keybindings in HelixTODO: document Helix-specific quirks (e.g. need to manually delete buffer after inlining a section)

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
Code Actions (Extract/Inline/AI/Transform)	Ctrl+. / Cmd+.	Right-click → "Quick Fix..."
Go to Definition (Follow Links)	F12	Right-click → "Go to Definition"
Find All References (Backlinks)	Shift+F12	Right-click → "Go to References"
Document Symbols (Table of Contents)	Ctrl+Shift+O / Cmd+Shift+O	Command Palette → "Go to Symbol"
Workspace Search (Global Search)	Ctrl+T / Cmd+T	Command Palette → "Go to Symbol in Workspace"
Format Document (Auto-Format)	Shift+Alt+F / Shift+Option+F	Right-click → "Format Document"
Rename Symbol (Rename File)	F2	Right-click → "Rename Symbol"

Additional VS Code Features

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

AI-Powered Actions

1. Select text you want to modify
2. Press `Ctrl+.` / `Cmd+.`
3. Choose from available AI 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:

```json
{
  "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 `iwes` 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
Code Actions	<leader>ca	Normal	Extract/Inline/AI/Transform code actions
Go to Definition	gd	Normal	Go to definition of symbol under cursor
Find References	gr	Normal	Find backlinks to current document
Document Symbols	<leader>ds	Normal	Navigate document outline
Workspace Search	<leader>ws	Normal	Global search with Telescope
Format Document	<leader>f	Normal/Visual	Auto-format document
Rename Symbol	<leader>rn	Normal	Rename symbol (including file & references)

LSP Keybindings

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

Telescope Integration

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

Configuration

Basic Setup

```lua
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

```lua
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:

```lua
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

AI-Powered Actions (if configured)

1. Select text in visual mode
2. Press `<leader>ca` to show code actions
3. Choose from available AI 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
<CR>	Open selected item
<C-x>	Open in horizontal split
<C-v>	Open in vertical split
<C-t>	Open in new tab
<C-u>	Scroll preview up
<C-d>	Scroll preview down
<C-q>	Send to quickfix list

Health Check

Run health checks to verify your setup:

```
:checkhealth iwe
```

This will check:

- IWE plugin installation
- `iwes` binary availability
- LSP server configuration
- Telescope integration
- IWE project configuration

Troubleshooting

Common Issues

1. **LSP Server Not Starting**

   ```
   # Check if iwes is in PATH
   which iwes

   # 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

Examples

Basic journal example

Lets take this Markdown journal as an example.

journal-2025.md

```markdown
# 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 *block-references*. This transformation maintains the document hierarchy while reducing level of details.

journal-2025.md

```markdown
# 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 support automated actions for graph transformations like this and it can just as easily reconstruct the **original** document buy 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 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 you 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

AI-Powered Contextual Commands

- **Configurable LLM integration**: Connect to any LLM provider with custom templates
- **Block-level AI actions**: Apply AI transformations to specific sections with full context awareness
- **Template-based prompts**: Customize AI behavior for different content types and use cases
- **Context-aware processing**: AI commands understand 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 Core Architecture documentation.

Detailed Comparisons

IWE vs markdown-oxide

markdown-oxide is a solid PKM LSP server focused on basic knowledge management:

Feature	IWE	markdown-oxide
Graph Transformations	✅ Extract/embed sections, convert lists↔sections, inline references	❌ Basic linking only
AI Integration	✅ Configurable LLM with contextual templates	❌ No AI features
Performance	✅ Rust-based, handles thousands of files instantly	✅ Good performance
Batch Operations	✅ CLI for bulk normalization and transformations	❌ LSP-only approach
Editor Support	✅ VSCode, Neovim, Zed, Helix	✅ VSCode, Neovim, Helix, Zed
Auto-formatting	✅ Comprehensive normalization on save	✅ Basic formatting
Daily Notes	❌ Not built-in	✅ Dedicated daily notes support
Backlinks	✅ Via graph processing	✅ Native backlink support

IWE's advantage: Advanced graph operations, AI 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:

Feature	IWE	Obsidian
Editor Integration	✅ Works with your preferred text editor	❌ Proprietary editor only
Cost	✅ Completely free and open source	⚠️ Free for personal use, $8/month for sync
Performance	✅ Rust-powered, instant operations	⚠️ Electron-based, can be slower
Graph Transformations	✅ Automated extract/embed operations	❌ Manual linking and organization
AI Integration	✅ Configurable LLM providers	⚠️ Limited, requires plugins
Collaboration	✅ Git-based, works with any VCS	⚠️ Requires paid Obsidian Sync

Feature	IWE	Obsidian
Cross-platform	✅ Consistent across all platforms	✅ Good cross-platform support
Graph Visualization	⚠️ CLI-based dot export	✅ Interactive graph view
Plugin Ecosystem	⚠️ Limited to LSP capabilities	✅ Rich plugin marketplace
Learning Curve	⚠️ Requires basic terminal knowledge	✅ GUI-friendly

IWE's advantage: Better for developers who want to stay in their preferred editor, need powerful automation, or want completely free sync via Git. Obsidian is better for users who prefer GUIs and interactive visualizations.

IWE vs zk.nvim/telekasten.nvim

zk.nvim and **telekasten.nvim** are Neovim-specific Zettelkasten solutions:

Feature	IWE	zk.nvim/telekasten
Editor Support	✅ VSCode, Neovim, Zed, Helix, others	❌ Neovim only
Graph Transformations	✅ Automated extract/embed, structural changes	❌ Basic note creation and linking
AI Integration	✅ Configurable LLM with templates	❌ Manual workflows only
Performance	✅ Rust-powered LSP	⚠️ Lua-based, editor-dependent
Batch Operations	✅ CLI for bulk operations	❌ One-note-at-a-time workflow
Auto-formatting	✅ Built-in normalization	❌ Requires external tools
Note Templates	✅ AI-powered dynamic templates	✅ Static templates
Search Integration	✅ LSP-based with any picker	✅ Telescope/fzf integration
Installation	✅ Single binary + editor extension	⚠️ Complex Neovim plugin setup

IWE's advantage: Works across all editors, provides powerful automation, and offers AI-enhanced workflows. zk.nvim/telekasten are better for Neovim purists who prefer simple, manual workflows.

IWE vs Denote/Emacs

Denote is a minimalist Emacs-based Zettelkasten system:

Feature	IWE	Denote/Emacs
Editor Support	✅ Cross-editor LSP support	❌ Emacs only
Simplicity	⚠️ More complex due to advanced features	✅ Extremely simple file-based approach
Graph Operations	✅ Automated transformations	❌ Manual file management
Performance	✅ Rust-powered	✅ Fast for basic operations

Feature	IWE	Denote/Emacs
AI Integration	✅ Built-in LLM support	❌ Would require custom Elisp
File Portability	✅ Standard markdown files	✅ Standard text/org files
Learning Curve	⚠️ LSP + terminal concepts	⚠️ Emacs + Elisp knowledge required
Extensibility	⚠️ Limited to domain model	✅ Unlimited Elisp customization
Database Dependency	❌ File-based like Denote	❌ File-based
Maintenance	✅ Automated normalization	⚠️ Manual organization required

IWE's advantage: Works with any editor, provides automation that Denote lacks, and includes AI capabilities. Denote is better for Emacs users who prefer extreme simplicity and unlimited customization.

Why Choose IWE?

IWE is the **only tool** that combines:

- 🚀 **Performance**: Rust-powered speed that handles thousands of files instantly
- 🤖 **Intelligence**: Integrated AI 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 need's 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.

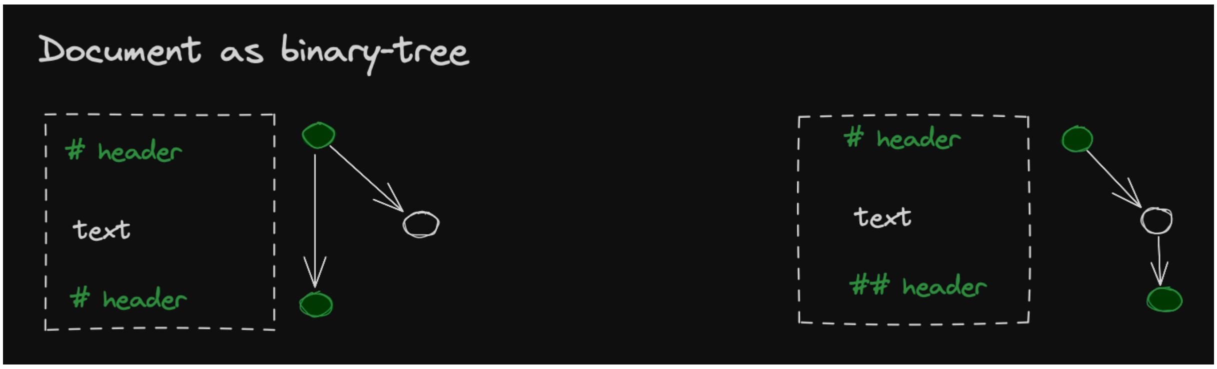
Core Architecture

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),       // Block references 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:

```rust
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:

```rust
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:

```rust
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:

```rust
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:

```rust
// 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:

```rust
// 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:

```rust
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:

```rust
// Normalization: Rewrite all documents with consistent formatting
fn normalize() { graph.export() → filesystem }

// Export: Generate visualization formats (DOT, etc.)
fn export() { graph → GraphData → DOTExporter }

// Contents: Generate table of contents
fn contents() { graph.paths() → filtered paths → markdown }

// 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:

```rust
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:

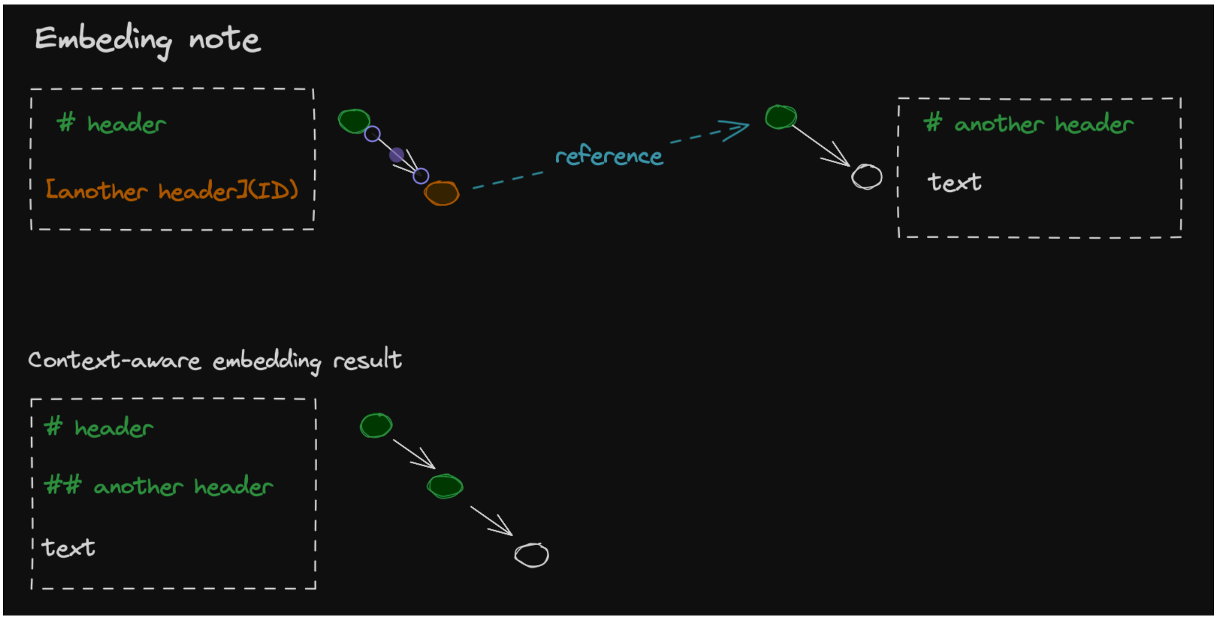
```
pub struct SearchPath {
    pub search_text: String,    // Concatenated plain text for matching
    pub node_rank: usize,      // Importance ranking
    pub key: Key,              // Source document
    pub root: bool,            // Is document root
    pub line: u32,             // Line number in 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:

```rust
// 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.