

Filesystem Retrieval Flow

1. Strategy Selection

The filesystem retriever treats the repository as a tree. Each node is either a directory or a file, and each candidate shown to the LLM contains only the node id, relative path, and type. Metadata fields are reserved for a later version and are not rendered in the active LLM-facing candidate payload.

The query entry point chooses the retrieval strategy in this order:

1. Read the tree metadata. If the tree source is not filesystem, dispatch to the document retriever.
2. If the caller passes `strategy="beam"`, use Beam directly. If the caller passes `strategy="block"`, use Block directly.
3. If the caller passes `strategy="auto"`, count tree nodes. Trees with at most 50 nodes use Beam; larger trees use Block.
4. Run the chosen retriever with the query, beam width, and final result limit.

The algorithms below show the filesystem path. Candidate order follows the stored filesystem order except for the optional post-merge path ordering in Algorithm 5.

2. Beam Retriever

Beam retrieval is level-by-level tree navigation. On each turn, every current beam is expanded to its direct children. The union of those children is the candidate set for that LLM call. If a beam is already a leaf, the leaf itself is kept as a candidate so it can be returned.

The Beam flow is implemented by Algorithm 1:

1. Initialize the beam set with the root node.
2. Expand every current beam into direct children.
3. Treat the union of expanded children as the current candidate set.
4. Ask the LLM to rank candidates and optionally signal `done`.
5. If the LLM says `done` but returned only directories, force `done=false` and continue exploring.
6. Add returned ids to the running top-candidate set.
7. Keep the top ranked expandable ids as the next beams.
8. Stop when k file results have been collected, the LLM says `done`, there are no candidates, or the depth limit is reached.

Algorithm 1 FilesystemBeam

Require: Tree id T , query q , beam width b , final result limit k

Ensure: Retrieved file nodes R

```
1:  $r \leftarrow \text{ROOT}(T)$ 
2:  $B \leftarrow [r]$ 
3:  $A \leftarrow []$ 
4:  $p \leftarrow \max(b, k)$ 
5: for  $t = 0, 1, \dots, \text{MAXDEPTH}(T)$  do
6:    $C \leftarrow []$ 
7:   for all  $u \in B$  do
8:      $Ch \leftarrow \text{CHILDREN}(T, u)$ 
9:     if  $Ch = \emptyset$  then
10:       $C \leftarrow C \cup \{u\}$ 
11:     else
12:       $C \leftarrow C \cup Ch$ 
13:     end if
14:   end for
15:   if  $C = \emptyset$  then
16:     break
17:   end if
18:    $(L, done) \leftarrow \text{LLMRANKBEAM}(q, C, A, p)$ 
19:   if  $done$  and no node in  $L$  is a file then
20:      $done \leftarrow \text{FALSE}$ 
21:   end if
22:    $Q \leftarrow [u \in L : u \text{ is a file}]$ 
23:    $A \leftarrow (A \cup Q)[1 : k]$ 
24:   if  $|A| \geq k$  then
25:     break
26:   end if
27:    $B \leftarrow []$ 
28:   for all  $u \in L$  do
29:     if  $u$  is a directory or expandable node then
30:       append  $u$  to  $B$ 
31:     end if
32:     if  $|B| = b$  then
33:       break
34:     end if
35:   end for
36:   if  $done$  then
37:     break
38:   end if
39: end for
40: return  $A[1 : k]$ 
```

3. Block Retriever

Block retrieval has the same recursive intent as Beam, but it does not present one large candidate list when the directory is wide. Instead, it partitions the current visible children into token-bounded blocks. For a block B_i , the block candidate set is exactly $C_i = B_i.\text{node_ids}$. The LLM returns a block-local ordered list from that block only; those returned ids are not inserted back into C_i . They are merged with other block results by order preservation, then split into file results and directory frontier nodes. When all entries in a block share a directory prefix, the rendered block may add `Path prefix: <prefix>` once and show each candidate path relative to that prefix.

The Block flow is implemented by Algorithms 2–5:

1. Initialize the frontier with the root, then collapse virtual single-child directory chains. This is handled by the main loop in Algorithm 2.
2. Materialize visible candidates from the current frontier. At turn 0 this is the top visible layer; later turns use the direct children of frontier nodes. This is handled by Algorithm 3.
3. Pack visible candidates into token-bounded blocks $\mathcal{B} = \{B_1, \dots, B_m\}$. This is handled by Algorithm 3.
4. Render each block using only id, path, and type. If a shared path prefix reduces tokens, render the prefix once and shorten candidate paths within that block. This is handled by Algorithm 3.
5. For each block B_i , define the allowed candidate set $C_i = B_i.\text{node_ids}$ and ask the LLM to rank only ids in C_i . This is handled by Algorithm 4.
6. Keep per-block outputs as block-local groups $\mathcal{G} = [L_1, \dots, L_m]$ and derive $M = \text{FLATTEN}(\mathcal{G})$ for the no-ranker path. This merge removes duplicates while preserving each block's LLM-returned order.
7. Optionally reorder across block groups without changing the relative order inside any L_i , then split the resulting list by node type: files enter the accumulated top-candidate result set, while directories enter the next frontier. This is handled by Algorithm 5.
8. If the LLM says `done` but the frontier still contains directories, override `done=false` and continue drilling down. This is handled by Algorithm 5.
9. Stop immediately once the accumulated file result set reaches the final result limit k .

Algorithm 2 FilesystemBlock Main Loop

Require: Tree id T , query q , beam width b , final result limit k , token budget τ

Ensure: Retrieved file nodes R

```

1:  $r \leftarrow \text{ROOT}(T)$ 
2:  $F \leftarrow \text{COLLAPSESINGLECHILDDIRS}([r])$ 
3:  $A \leftarrow []$ 
4:  $P \leftarrow []$ 
5:  $\hat{b} \leftarrow b$  if set, otherwise 1
6:  $p \leftarrow \max(\hat{b}, k)$  ▷ LLM pick limit, not the final result limit
7:  $t \leftarrow 0$ 
8: while  $t < \text{MAXROUNDS}$  do
9:   if  $t > 0$  and no node in  $F$  has children then
10:     break
11:   end if
12:    $(\mathcal{B}, X) \leftarrow \text{BUILDBLOCKCANDIDATES}(T, F, t, \tau)$ 
13:   if  $X = \emptyset$  or  $\mathcal{B} = \emptyset$  then
14:     break
15:   end if
16:    $p_t \leftarrow p$  if  $|\mathcal{B}| = 1$ , otherwise  $\max(p, 2)$ 
17:    $(\mathcal{G}, M, \text{done}) \leftarrow \text{CALLBLOCKSANDMERGE}(q, \mathcal{B}, F, P, p_t)$ 
18:    $(F, P, A, \text{done}) \leftarrow \text{UPDATESTATE}(T, q, \mathcal{G}, M, \text{done}, b, k, A)$ 
19:   if  $|A| \geq k$  or  $\text{done}$  then
20:     break
21:   end if
22:    $t \leftarrow t + 1$ 
23: end while
24: return  $A[1 : k]$ 

```

Algorithm 3 BuildBlockCandidates

Require: Tree id T , frontier F , turn t , token budget τ

Ensure: Token-bounded blocks \mathcal{B} and visible candidates X

```
1: if  $t = 0$  then
2:    $X \leftarrow \text{TOPVISIBLENODES}(T, F_1)$ 
3: else
4:    $X \leftarrow []$ 
5:   for all  $u \in F$  do
6:      $X \leftarrow X \cup \text{CHILDREN}(T, u)$ 
7:   end for
8: end if
9: if  $X = \emptyset$  then
10:  return  $(\emptyset, \emptyset)$ 
11: end if
12: Render each candidate using id, path, and type
13: Render a shared Path prefix when it shortens the block
14:  $\mathcal{B} \leftarrow \text{PACKINTOBLOCKS}(X, \tau)$ 
15: return  $(\mathcal{B}, X)$ 
```

Algorithm 4 CallBlocksAndMerge

Require: Query q , blocks \mathcal{B} , frontier F , previous top candidates P , LLM pick limit p

Ensure: Block-local id groups \mathcal{G} , merged ids M , and stop signal $done$

```
1: for all  $i = 1, \dots, |\mathcal{B}|$  in parallel do
2:    $B_i \leftarrow \mathcal{B}_i$ 
3:    $C_i \leftarrow B_i.\text{node\_ids}$ 
4:    $(L_i, d_i) \leftarrow \text{LLMRANKBLOCK}(q, B_i, C_i, F, P, p)$ 
5: end for
6:  $\mathcal{G} \leftarrow []$ 
7: for  $i = 1, \dots, |\mathcal{B}|$  do
8:    $\mathcal{G} \leftarrow \mathcal{G} \parallel [L_i]$ 
9: end for
10:  $\mathcal{G} \leftarrow \text{UNIQUEPRESERVEBLOCKORDER}(\mathcal{G})$ 
11:  $M \leftarrow \text{FLATTEN}(\mathcal{G})$ 
12:  $done \leftarrow \text{ALLTRUE}(\{d_i\})$ 
13: return  $(\mathcal{G}, M, done)$ 
```

Algorithm 5 UpdateState

Require: Tree id T , query q , block-local id groups \mathcal{G} , merged ids M , stop signal $done$, beam width b , final result limit k , top candidates A

Ensure: Next frontier F , previous top candidates P , top candidates A , stop signal $done$

```
1:  $M' \leftarrow \text{CONSTRAINEDPATHORDER}(\mathcal{G}, q)$ 
2:  $Q \leftarrow [u \in M' : u \text{ is a file}]$ 
3:  $D \leftarrow [u \in M' : u \text{ is a directory}]$ 
4:  $r \leftarrow \max(0, k - |A|)$ 
5:  $A \leftarrow A \cup Q[1 : r]$ 
6:  $P \leftarrow A[1 : k]$ 
7: if  $|A| \geq k$  then
8:   return  $(\emptyset, P, A[1 : k], \text{TRUE})$ 
9: end if
10:  $F' \leftarrow D[1 : b]$ 
11:  $F \leftarrow \text{COLLAPSESINGLECHILDDIRS}(F')$ 
12: if  $done$  and some node in  $F$  still has children then
13:    $done \leftarrow \text{FALSE}$ 
14: end if
15: return  $(F, P, A[1 : k], done)$ 
```

Block semantics. For every block call, the LLM can only select ids from the current block’s allowed list. If the current directory produces three blocks B_1, B_2, B_3 , the calls produce L_1, L_2, L_3 . The algorithm keeps these as block-local groups $\mathcal{G} = [L_1, L_2, L_3]$ and derives $M = \text{FLATTEN}(\mathcal{G})$. This is a merge, not another LLM ranking step. M is therefore called `merged_ids`, not a ranked list. Without path ordering, its order is inherited from block order and each block’s local LLM order. With optional path ordering, only cross-block order may change; the relative order inside each L_i is fixed. Files enter `top_candidate_ids`; directories enter the next frontier after beam-width truncation. The next candidate set is produced only by expanding that frontier.

Block cache method. Block retrieval separates stable tree text from dynamic query state. The stable part is the rendered block content; the dynamic part contains the query, current exploration path, previous top candidates, and the ids allowed for the current block call. If cache is enabled, the retriever builds a cache payload from a static instruction segment plus a bounded cache window of previously useful block segments. The current block is also placed in the cache payload when `cache_current_block` is enabled at the top level, or when `cache_subtree_block` is enabled in recursive subtree rounds.

After each round, the blocks that contain the next frontier are appended to the cache window. The first top-level block may be pinned, while later entries can be trimmed when the cache-window token budget is exceeded. The LLM call then sends the cache payload as cached content and sends only the query-specific message as the dynamic user message. If the LLM adapter does not support cached calls, the same content is concatenated into a normal prompt; retrieval semantics are the same, but no provider-side cache saving is used. The `Path prefix` line is separate from this cache mechanism: it is only a token-compression rendering trick inside a block.

Path ordering. The deterministic merge preserves block order and block-local order. It does not compare candidates from different blocks. When a ranker is supplied, the path scorer runs once over the block-local groups $\mathcal{G} = [L_1, \dots, L_m]$ and before the file/directory split:

$$M' = \text{CONSTRAINEDPATHORDER}(\mathcal{G}, q).$$

Only the current head of each group may be selected at each step, so different blocks can be interleaved while the relative order inside every L_i remains unchanged. Without a ranker, $M' = \text{FLATTEN}(\mathcal{G})$.

The optional scorer is path-only:

$$s(u, q) = 3 \text{BM25}(q, \text{basename}(u)) + 1.5 \text{BM25}(q, \text{parent}(u)) + \text{BM25}(q, \text{path}(u)).$$

The ordered list M' is then split: files fill the remaining $k - |A|$ result slots, and directories are truncated to the beam width for the next frontier.

4. Output Semantics

In filesystem mode, `candidate_set`, `ranked_ids`, `frontier`, and `top_candidate_ids` have different meanings. The candidate set is the current block-local pool shown to the LLM. `ranked_ids` is only the LLM-returned order inside one block and may contain files or directories. After deterministic merge, the cross-block list is `merged_ids`; it is not a global ranking. If optional path ordering is enabled, the retriever derives one ordered list from `merged_ids` before splitting it. Files then enter `top_candidate_ids`, which is the maintained result set, and directories form the next frontier. Final filesystem output returns at most k entries from `top_candidate_ids` and their contents. The separate `pick_limit` controls how many ids an LLM call may return inside one round; it is not the final result count.

Appendix A. LLM Prompt Payloads

A.1 Tool schema

Both retrievers use the same tool-call contract. The LLM must return one rank tool call.

Tool schema

```
TOOLS = [  
  {  
    "name": "rank",  
    "description": "Rank candidate node ids for the query",  
    "input_schema": {  
      "type": "object",  
      "properties": {  
        "ranked_ids": {  
          "type": "array",  
          "items": {"type": "string"}  
        },  
        "done": {"type": "boolean"},  
      },  
      "required": ["ranked_ids"],  
    },  
  },  
]
```

A.2 Rendered prompt examples

The boxes below show examples after the templates are rendered for an LLM call. Angle-bracket fields are filled from the current tree state before each call.

A.2.1 Beam prompt

Source template: contextdb/prompts/beam_fs.jinja

```
You are navigating a source code repository to find files relevant to this query:  
  
Query: <query>  
  
Already top candidates:  
- <previous top candidate node id>  
  
Current candidates (pick up to <pick_limit>, most relevant first):  
- id: <node id>  
  path: <relative path>  
  type: <directory | file>  
- id: <node id>  
  path: <relative path>  
  type: <directory | file>  
  
RULES:  
- Select directories to explore deeper, or files if they match the query  
- set done=true ONLY when you've found specific source files that answer the query  
- set done=false if you returned directories that need further exploration  
  
Return ONE tool call "rank" with:  
- ranked_ids: list of candidate ids in best-to-worst order  
- done: true or false
```

A.2.2 Block prompt

Source templates: block_fs_cache_prefix.jinja and block_fs.jinja. The first part is the cached block prefix; the second part is the dynamic user message for the current query and allowed ids.

Cached block prefix

```
=== REPOSITORY DIRECTORY TREE ===
```

```
Path prefix: <shared directory prefix>/
```

```
- id: n1
  path: <path relative to prefix>
  type: <directory | file>
- id: n2
  path: <path relative to prefix>
  type: <directory | file>
```

You are navigating a source code repository's directory tree to find files relevant to a query. Each entry above has:

```
- id: a unique node identifier
- path: relative file/directory path
- type: file or directory
```

RULES:

```
- Select directories to drill deeper into, or files if they directly match the query
- set done=false if returned directories need further exploration
- Select ONLY ids from the allowed list provided below
```

Return ONE tool call "rank" with:

```
- ranked_ids: list of candidate ids in best-to-worst order
- done: true ONLY when the returned ids are specific source files
```

Dynamic message

```
Query: <query>
```

```
Top candidates so far:
```

```
- <previous top candidate path>
```

```
Current exploration path:
```

```
- <current frontier path>
```

The LAST BLOCK above is the CURRENT BLOCK.
IDs in each block are local to that block.
Return only ids from the CURRENT BLOCK.

```
Selectable ids in the CURRENT BLOCK:
```

```
- n1
- n2
```

```
Pick up to <pick_limit> candidates most relevant to the query, best first.
```

Expected tool call

```
rank({
  "ranked_ids": ["n1"],
  "done": true
})
```

The implementation maps local aliases such as n1 back to real node ids before splitting ranked ids into the top-candidate result set and the next frontier.