FileContextService

September 4, 2026

FileContextService

Kind: Service

Source: `atloria-monorepo/apps/api/src/ai/services/file-context.service.ts`

Optimized file context service inspired by Claude Code & Cursor

Key optimizations:

1. Lazy loading - Only read files when needed
2. Chunking - Read files in segments (line ranges)
3. Token counting - Estimate tokens before sending
4. Caching - Cache file contents with invalidation
5. Smart filtering - Skip irrelevant files (node_modules, binaries, etc.)
6. Relevance ranking - Prioritize most relevant files

`FileContextService` prepares efficient repository context for AI-assisted workflows in the API. It lazily reads and caches files, filters irrelevant paths, ranks relevant files, and builds token-budgeted chunks that can be formatted for an AI prompt.

Methods

Method	Signature	Returns	Description
`getFileMetadata`	`getFileMetadata(filePath: string)`	`Promise<FileMetadata>`	Get file metadata without reading entire content
`readFile`	`readFile(filePath: string)`	`Promise<string>`	Read file with caching

Method	Signature	Returns	Description
`readFileChunk`	`readFileChunk(filePath: string, startLine: number, endLine: number)`	`Promise<FileChunk>`	Read specific line range from file (chunking optimization)
`findRelevantFiles`	`findRelevantFiles(baseDir: string, patterns: string[], options: {		

```
  maxFiles?: number;
  maxTokens?: number;
  exclude?: string[];
})` | `Promise<FileMetadata[]>` | Find relevant files based on glob patterns Automatically filt
```

`buildContext`	`buildContext(files: string[], budget: number, keywords: string[])`	`Promise<{ chunks: FileChunk[]; budget: ContextBudget; skippedFiles: string[]; }>`	Build optimized context from multiple files Returns chunks sorted by relevance with token budget management
`estimateTokens`	`estimateTokens(content: string)`	`number`	Estimate token count for content Uses simple heuristic: 1 token ≈ 4 characters for code
`clearCache`	`clearCache(filePath: string)`	`void`	Clear file cache (useful for testing or when files change)
`getCacheStats`	`getCacheStats()`	`unknown`	Get cache statistics
`formatChunksForAI`	`formatChunksForAI(chunks: FileChunk[], baseDir: string)`	`string`	Format chunks for AI context

Where it refuses work

- `FileContextService` stops the work with an early return when `matchedLines.size === 0`.

When something fails

- `FileContextService` handles failure in 4 places: it lets it reach the caller in 2, logs it and continues in 1, and turns it into a return value in 1.

Diagram

```
sequenceDiagram
  participant Client as AI Workflow / Caller
  participant Service as FileContextService
  participant FS as File System
  participant Cache as File Cache

  Client->>Service: buildContext(query, filePaths, tokenBudget)
  Service->>Service: findRelevantFiles()
  Service->>Service: Filter ignored/binary files
  Service->>Service: Rank files by relevance

  loop Relevant files within budget
    Service->>Cache: Check cached metadata/content
    alt Cache miss or invalidated
      Service->>FS: getFileMetadata() / readFileChunk()
      FS-->>Service: File metadata and chunk content
      Service->>Cache: Store cached result
    else Cache hit
      Cache-->>Service: Cached metadata/content
    end
    Service->>Service: estimateTokens(chunk)
  end

  Service-->>Client: chunks, budget, skippedFiles
  Client->>Service: formatChunksForAI(chunks)
  Service-->>Client: Prompt-ready file context
```

Usage

```
import { FileContextService } from './ai/services/file-context.service';

// In NestJS, prefer injecting FileContextService through a constructor.
async function buildAiPromptContext(
  fileContextService: FileContextService,
  userRequest: string,
  candidateFiles: string[],
) {
  const context = await fileContextService.buildContext(
    userRequest,
    candidateFiles,
    12_000, // token budget for repository context
  );

  const formattedContext = fileContextService.formatChunksForAI(
    context.chunks,
  );

  return {
    promptContext: formattedContext,
```

```
    skippedFiles: context.skippedFiles,
    tokenUsage: context.budget,
  };
}

// Read only a targeted portion of a file when full context is unnecessary.
async function inspectController(
  fileContextService: FileContextService,
) {
  const chunk = await fileContextService.readFileChunk(
    'src/users/users.controller.ts',
    1,
    150,
  );

  return {
    content: chunk.content,
    estimatedTokens: fileContextService.estimateTokens(chunk.content),
  };
}
```

AI Coding Instructions

- Use `buildContext()` for prompt preparation instead of reading every candidate file manually; it applies relevance ranking and token-budget enforcement.
- Prefer `readFileChunk()` for targeted inspection of large files, and reserve `readFile()` for files that genuinely require complete content.
- Pass repository-relative paths and expect irrelevant directories or binary files to be excluded automatically; do not bypass filtering for `node_modules` or generated assets.
- Reuse the service instance so its cache can reduce filesystem reads, and call `clearCache()` only when repository state changes or deterministic fresh reads are required.
- Use `formatChunksForAI()` when constructing model prompts so file paths, line ranges, and chunk boundaries remain clear to the AI.

Referenced By

- `AIModule` (MODULE_PROVIDES)
- `AIModule` (MODULE_EXPORTS)
- `CodeAnalysisService` (DEPENDS_ON)
- `UIAnalyzerService` (DEPENDS_ON)