CodeAnalysisService

September 4, 2026

CodeAnalysisService

Kind: Service

Source: `atloria-monorepo/apps/api/src/ai/services/code-analysis.service.ts`

Code Analysis Service using optimized file context

Implements Claude Code & Cursor style optimizations:

- Lazy loading of files
- Token budget management
- Relevance-based file selection
- Chunked reading for large codebases

`CodeAnalysisService` is a NestJS service that analyzes source code using an optimized “file context” approach similar to Claude Code/Cursor. It lazily loads and chunks files, manages an explicit token budget, and selects the most relevant files to include in the analysis context. This service typically sits behind API endpoints or other AI orchestration services to prepare high-signal code context for LLM prompts.

Methods

Method	Signature	Returns	Description
`analyzeCodebase`	`analyzeCodebase(options: {		

```
baseDir: string;
patterns: string[];
query: string;
tokenBudget?: number;
keywords?: string[];
provider?: string;
```

}) | Promise | Analyze code in a directory with optimized context loading Example usage: ```typescript const result = await analyzeCodebase({ baseDir: '/path/to/project', p… |
| analyzeFiles | analyzeFiles(options: {
files: Array<{ path: string; lines?: [number, number] }>;
query: string;
provider?: string;
}) | Promise | Analyze specific files with line-level precision Example usage: ```typescript const result = await analyzeFiles({ files: [{ path: 'src/auth/auth.service.ts'… |
| streamCodebaseAnalysis | streamCodebaseAnalysis(options: {
baseDir: string;
patterns: string[];
query: string;
tokenBudget?: number;
keywords?: string[];
provider?: string;
}) | AsyncIterableIterator | Stream code analysis (for large responses) |
| getCacheStats | getCacheStats() | unknown | Get cache statistics |
| clearCache | clearCache(filePath: string) | unknown` | Clear file cache |

Dependencies

- `FileContextService`
- `AIService`

Diagram

```
sequenceDiagram
  autonumber
  participant Caller as API/Orchestrator
  participant CAS as CodeAnalysisService
  participant FS as File System/Repo
  participant TB as Token Budget Manager
  participant RS as Relevance Selector
  participant CH as Chunk Reader

  Caller->>CAS: analyze(query, repoRoot, options)
  CAS->>RS: rankCandidateFiles(query, repoIndex)
  RS-->>CAS: relevantFiles[]
  CAS->>TB: init(budget, model)
  TB-->>CAS: remainingTokens

  loop for each relevant file (until budget exhausted)
    CAS->>FS: stat/read metadata (lazy)
```

```
    FS-->>CAS: size/mtime
    alt small file
      CAS->>FS: readFile(path)
      FS-->>CAS: content
    else large file
      CAS->>CH: readChunks(path, chunkSize)
      CH->>FS: stream/read ranges
      FS-->>CH: chunkContent
      CH-->>CAS: chunkContent
    end
    CAS->>TB: estimateTokens(content)
    TB-->>CAS: allow/trim/stop
  end

  CAS-->>Caller: analysisContext (selected files + excerpts)
```

Usage

```ts
import { Injectable } from '@nestjs/common';
import { CodeAnalysisService } from './ai/services/code-analysis.service';

@Injectable()
export class AiOrchestrationService {
  constructor(private readonly codeAnalysis: CodeAnalysisService) {}

  async buildPromptContext() {
    const repoRoot = process.cwd();
    const query = 'Find where authentication is enforced and how JWT is validated';

    // Options are illustrative; use the service's actual option names/types.
    const context = await this.codeAnalysis.analyze(query, repoRoot, {
      tokenBudget: 12000,
      maxFiles: 20,
      chunkSize: 4000,
      includePatterns: ['apps/api/src/**'],
      excludePatterns: ['**/dist/**', '**/node_modules/**'],
    });

    // `context` can then be appended to an LLM prompt.
    return {
      system: 'You are a codebase assistant.',
      user: `Question: ${query}\n\nCode Context:\n${context}`,
    };
  }
}
```

AI Coding Instructions

- Preserve the optimization pipeline: relevance ranking → lazy loading → chunked reads → token-budget enforcement; avoid eager full-repo reads.
- When modifying file selection, ensure exclude/include patterns are applied before reading content to prevent wasted IO and token usage.
- Be careful with token estimation: always budget using the final text actually sent (post-trim/post-chunk) to avoid prompt overflow.
- Integration point: return a deterministic, ordered context (stable sorting/ranking) so repeated runs produce similar prompts and diff-friendly outputs.
- For large files, prefer excerpting relevant regions over adding more files; adding too many low-signal files is a common quality pitfall.

Relationships

- DEPENDS_ON → `FileContextService`
- DEPENDS_ON → `AIService`

Referenced By

- `AIModule` (MODULE_PROVIDES)
- `AIModule` (MODULE_EXPORTS)