

Language Server Protocol (LSP) Server: Design Document

Overview

This system implements a Language Server Protocol server that provides IDE features like completion, hover, and diagnostics by maintaining document state, parsing code into abstract syntax trees, and responding to JSON-RPC requests from language clients. The key architectural challenge is building a responsive, stateful server that can efficiently track document changes and provide real-time language analysis without blocking the editor.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

Milestone(s): This section provides essential background for all milestones, particularly Milestone 1 (JSON-RPC & Initialization) and the overall architectural understanding needed throughout the project.

The IDE Integration Problem

Think of building language support for every editor as trying to create a universal translator for a United Nations meeting. Without a common protocol, each language team would need to build custom interpreters for every editor platform, and each editor would need custom plugins for every programming language. This creates an **N×M integration problem** where N languages multiplied by M editors results in an exponential explosion of integration work.

The Language Server Protocol (LSP) solves this by establishing a standardized communication protocol between language tools and development environments. Instead of building N×M integrations, we now build N language servers and M language clients, reducing the complexity to N+M integrations. This is analogous to how HTTP became the universal protocol for web communication—once established, any browser can talk to any web server using the same standardized message format.

Before LSP's introduction by Microsoft in 2016, the developer tools ecosystem faced several fundamental challenges that made building comprehensive language support prohibitively expensive for most programming languages:

Duplication of Effort Across Editors

Every editor or IDE that wanted to support a programming language had to implement language-specific features independently. Consider TypeScript support: before LSP, the TypeScript team had to build separate plugins for Visual Studio Code, Sublime Text, Vim, Emacs, Atom, WebStorm, and dozens of other editors. Each plugin

required understanding of the editor's extension API, learning its threading model, adapting to its user interface conventions, and maintaining compatibility across editor versions.

This duplication meant that popular languages like JavaScript, Python, and Java received excellent tooling support across multiple editors because companies could justify the investment. However, smaller or domain-specific languages often had minimal IDE support because the cost of building and maintaining plugins for every major editor was prohibitive.

Inconsistent Feature Sets

Even when multiple editors supported the same language, the feature sets were inconsistent. The Go plugin for VS Code might provide excellent auto-completion and refactoring tools, while the Go plugin for Sublime Text only offered basic syntax highlighting and error detection. This inconsistency created friction for developers who wanted to switch editors or work in teams with diverse tooling preferences.

Resource Intensive Language Processing

Language analysis is computationally expensive. Parsing large codebases, building symbol tables, performing type checking, and maintaining cross-file dependencies requires significant CPU and memory resources. When each editor plugin implemented its own language processing engine, these resources were duplicated across tools. A developer running VS Code, a terminal with Vim, and a debugging session might have three separate TypeScript analysis engines consuming system resources simultaneously.

Maintenance Burden

Language tool maintainers faced an enormous maintenance burden. When the language specification changed, when new features were added, or when bugs were discovered in the analysis engine, fixes had to be propagated across every editor plugin. Each plugin had different release cycles, testing procedures, and compatibility requirements, making coordinated updates nearly impossible.

Limited Innovation in Language Tooling

The high barrier to entry for building language tools meant that innovative language features often lacked proper IDE support. New programming languages struggled to gain adoption because developers expected modern IDE features like intelligent auto-completion, real-time error detection, and refactoring support. Building these features for multiple editors was often beyond the resources of language creators, creating a chicken-and-egg problem where languages couldn't grow without tooling, but tooling wasn't worth building without adoption.

LSP addresses these problems by establishing a **standardized JSON-RPC protocol** for communication between language servers and editor clients. The protocol defines precise message formats for common IDE features like completion suggestions, hover information, go-to-definition, find-references, and diagnostic reporting.

Here's how the architecture transforms the integration landscape:

Aspect	Pre-LSP Approach	LSP Approach	Benefit
Integration Complexity	$N \text{ languages} \times M \text{ editors} = N \times M \text{ plugins}$	$N \text{ language servers} + M \text{ LSP clients} = N+M \text{ integrations}$	Quadratic to linear complexity reduction
Feature Consistency	Each plugin implements features differently	All editors get identical language features from the same server	Uniform experience across editors
Resource Usage	Each editor runs separate analysis engines	Single language server serves multiple editor clients	Reduced memory and CPU consumption
Maintenance Overhead	Bug fixes require updates to $N \times M$ plugins	Bug fixes only require updating N language servers	Dramatically simplified maintenance
Development Investment	Language teams must learn M different editor APIs	Language teams implement one LSP-compliant server	Lower barrier to building language tools

The protocol operates through a **client-server architecture** where the language server runs as a separate process, maintaining the authoritative state of the codebase and performing all language analysis. Editor clients communicate with the server through JSON-RPC messages, requesting language services and receiving structured responses.

This separation of concerns provides several architectural advantages. The language server can be implemented in the most appropriate language for the domain—a C++ language server for optimal performance, a JavaScript language server for ecosystem integration, or a Rust language server for memory safety. Meanwhile, editor clients only need to implement the LSP protocol once to support dozens of programming languages.

Key Insight: LSP's success stems from recognizing that language analysis and editor integration are fundamentally different problems that benefit from separation. Language analysis requires deep domain knowledge and computational resources, while editor integration requires understanding user interface patterns and development workflows.

The protocol also enables **advanced deployment scenarios** that were difficult with traditional plugins. Language servers can run remotely, enabling cloud-based development environments where the analysis engine runs on powerful server hardware while the editor runs on lightweight client devices. Multiple developers can share a single language server instance for collaborative editing scenarios. Language servers can be containerized and deployed as microservices in development infrastructure.

Existing Approaches

Before diving into our LSP server implementation, it's valuable to understand the evolution of language tooling approaches and why LSP emerged as the preferred solution. Each approach represents different trade-offs between integration complexity, performance, consistency, and development effort.

Traditional Editor Plugins

The first generation of language support relied on **editor-specific plugins** written in the editor's extension language or API. Vim plugins were written in Vimscript, Emacs packages in Emacs Lisp, VS Code extensions in JavaScript/TypeScript, and Sublime Text plugins in Python.

Think of this approach like building custom adapters for every possible device combination—each plugin is a bespoke integration that speaks the editor's native language but implements language analysis from scratch.

Aspect	Description	Trade-offs
Architecture	Plugin code runs in editor process, directly calling editor APIs	Tight integration but high coupling
Language Processing	Each plugin implements parsing, symbol resolution, and analysis independently	Full control but massive duplication
Performance	Direct memory access and API calls provide low latency	Fast response but resource duplication
Maintenance	Each plugin requires updates when language or editor changes	Immediate updates but N×M maintenance burden
Distribution	Plugins distributed through editor-specific package managers	Native distribution but fragmented ecosystem

Traditional plugins excelled in scenarios where deep editor integration was crucial. A Vim plugin could leverage Vim's powerful text manipulation primitives, while an Emacs package could integrate with Emacs's buffer management and key binding systems. However, this tight coupling created significant challenges:

The **knowledge barrier** was substantial. Building a quality TypeScript plugin for Vim required expertise in both TypeScript language internals and Vim's extension architecture. Few developers possessed both skill sets, leading to plugins of varying quality and completeness.

The **Version synchronization** became a nightmare as projects grew. When TypeScript released version 4.0 with new syntax features, each plugin maintainer had to independently implement support for the new grammar, update their parser, and test across different editor versions. This process could take months, creating inconsistent support across the ecosystem.

The **Resource consumption** was inefficient. A developer using multiple editors might have several JavaScript analysis engines running simultaneously, each parsing the same codebase and maintaining duplicate symbol tables in memory.

Embedded Language Interpreters

The second approach embedded the language's own runtime or analysis tools directly into the editor process. This pattern was popular for dynamic languages with accessible runtime environments.

Consider this like embedding a specialized calculator into a general-purpose computer—instead of implementing mathematical functions from scratch, you embed a component that already knows how to perform complex calculations.

Aspect	Description	Trade-offs
Architecture	Editor hosts language runtime (Python interpreter, Node.js, etc.)	Authoritative analysis but complex embedding
Language Processing	Leverage existing language tools (AST parsers, type checkers, linters)	Reuse existing tools but integration complexity
Performance	Runtime startup cost but authentic language semantics	Slower initialization but accurate results
Maintenance	Language tool updates automatically improve editor support	Reduced maintenance but version compatibility issues
Distribution	Requires language runtime installation on developer machines	Authentic tooling but deployment complexity

This approach was particularly successful for languages with rich runtime reflection capabilities. Python's `jedi` library could provide excellent completion suggestions by actually importing modules and introspecting their structure. JavaScript tools like `tern.js` could analyze code using the same V8 engine that would execute it.

However, embedded interpreters introduced their own challenges:

Process stability became a concern when language runtimes crashed or consumed excessive resources. A poorly written Python script could freeze the entire editor if the embedded interpreter hung during analysis.

Security implications emerged when editors began executing user code for analysis purposes. Malicious code could potentially access editor internals or user data through the embedded runtime environment.

Version conflicts arose when projects used different language versions than the embedded interpreter. A Python project targeting Python 3.9 might receive incorrect analysis from an editor running Python 3.7.

Protocol-Based Solutions (Pre-LSP)

Before LSP standardized the approach, several projects experimented with custom protocols for editor-language communication. Each created bespoke message formats and communication mechanisms.

Think of this era like the early days of instant messaging—every company (AIM, MSN, Yahoo) had their own protocol, requiring users to run multiple clients to communicate with everyone.

Protocol/Tool	Editor Support	Language Focus	Protocol Design
Language Server Index Format (LSIF)	Multi-editor	Multi-language	Dump-based, pre-computed indices
Eclim	Vim, Emacs	Java	Eclipse JDT backend with custom protocol
OmniSharp	Multi-editor	C#	HTTP/JSON-RPC hybrid protocol
Racer	Multi-editor	Rust	Custom line-based protocol
Tern	Multi-editor	JavaScript	JSON-RPC over HTTP

These early protocol-based solutions demonstrated the viability of separating language analysis from editor integration, but each required custom client implementations. A developer who wanted Rust support in both Vim and VS Code had to install different plugins that spoke different protocols to different server implementations.

Comparison of Approaches

Let's evaluate these approaches across key dimensions that matter for language tool development:

Decision: Protocol-Based Architecture

- **Context:** Need to provide consistent language support across multiple editors while minimizing development and maintenance overhead
- **Options Considered:**
 1. Traditional editor plugins with embedded analysis
 2. Embedded language interpreters with editor bindings
 3. Custom protocol-based language servers
 4. Standardized protocol (LSP) with language servers
- **Decision:** Implement a Language Server Protocol compliant server
- **Rationale:** LSP provides the optimal balance of consistency, maintainability, and ecosystem support. The protocol is mature, well-documented, and supported by major editors. Building an LSP server allows us to support VS Code, Vim/Neovim, Emacs, Sublime Text, and other editors with a single implementation.
- **Consequences:** We gain broad editor compatibility and can leverage existing LSP client implementations, but we must conform to LSP's request/response model and may not be able to implement editor-specific optimizations.

Evaluation Criteria	Editor Plugins	Embedded Interpreters	Custom Protocols	LSP
Development Effort	High ($N \times M$ implementations)	Medium (M integrations)	Medium (M client implementations)	Low (Single server implementation)
Feature Consistency	Low (each plugin differs)	Medium (runtime dependent)	Medium (protocol dependent)	High (standardized protocol)
Performance	High (native integration)	Medium (runtime overhead)	High (optimized communication)	High (optimized JSON-RPC)
Maintenance Burden	Very High ($N \times M$ combinations)	High (M runtime integrations)	Medium (M client protocols)	Low (Single server + client reuse)
Ecosystem Support	Editor-specific	Language-specific	Tool-specific	Universal
Resource Efficiency	Low (duplication)	Low (embedded runtimes)	High (shared servers)	High (shared servers)
Innovation Velocity	Slow (distributed updates)	Medium (centralized tools)	Medium (server updates)	Fast (centralized server)

The LSP approach emerges as the clear winner for most scenarios, which explains its rapid adoption across the development tools ecosystem. By implementing an LSP-compliant server, we can provide high-quality language support to developers regardless of their editor choice while maintaining a single codebase.

⚠ Pitfall: Protocol Complexity Underestimation

Many developers underestimate the complexity of implementing LSP correctly. The protocol specification is comprehensive, covering dozens of message types, capability negotiation, document synchronization, and error handling scenarios. A common mistake is implementing only the basic completion and hover features without proper initialization handshake, lifecycle management, or incremental document updates. This results in a server that works for simple scenarios but fails in real-world usage with multiple files, concurrent requests, or editor restarts. To avoid this, study the complete LSP specification and implement the protocol systematically, starting with the foundational transport and initialization layers before adding language features.

Our LSP server implementation will demonstrate how to build a robust, protocol-compliant language server that provides modern IDE features while maintaining the architectural benefits of the protocol-based approach. The next sections will detail the specific design decisions and implementation strategies that enable this capability.

Implementation Guidance

Building an LSP server requires understanding both the protocol mechanics and the broader ecosystem of language tooling. This section provides concrete guidance for starting your implementation with a solid

foundation.

Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
JSON-RPC Transport	Native stdio with manual framing	<code>@microsoft/vscode-jsonrpc</code> library	Manual implementation teaches protocol details; library handles edge cases
Message Parsing	Built-in <code>JSON.parse/stringify</code>	Schema validation with <code>ajv</code> or <code>zod</code>	Built-in parsing is sufficient for learning; validation helps with debugging
Document Storage	In-memory <code>Map<URI, DocumentState></code>	Persistent storage with <code>sqlite3</code>	Memory storage is simpler and sufficient for most use cases
AST Parsing	Language-specific parser (e.g., <code>@babel/parser</code> for JS)	Universal parser with Tree-sitter	Language-specific parsers provide better semantics; Tree-sitter offers consistency
Concurrency Model	Async/await with Promise queues	Worker threads for CPU-intensive parsing	Single-threaded async is easier to debug and sufficient for moderate workloads

For learning purposes, start with the simple options to understand the fundamental concepts, then migrate to advanced options as your requirements grow.

Recommended File Structure

Organize your LSP server codebase to separate protocol concerns from language analysis:

```

lsp-server/
├── src/
│   ├── main.ts           ← Entry point and server lifecycle
│   ├── transport/
│   │   ├── jsonrpc.ts    ← JSON-RPC message framing and parsing
│   │   ├── stdio-transport.ts    ← stdin/stdout communication
│   │   └── message-types.ts    ← LSP message interfaces
│   ├── protocol/
│   │   ├── lsp-server.ts    ← Main LSP protocol handler
│   │   ├── initialization.ts    ← Initialize/initialized handshake
│   │   ├── capabilities.ts    ← Server capabilities negotiation
│   │   └── lifecycle.ts      ← Shutdown/exit handling
│   ├── document/
│   │   ├── document-manager.ts    ← Document synchronization and storage
│   │   ├── text-document.ts    ← Document representation and versioning
│   │   └── position-utils.ts    ← Position/Range calculation utilities
│   ├── language/
│   │   ├── parser.ts          ← AST parsing and caching
│   │   ├── symbol-resolver.ts    ← Symbol table and scope analysis
│   │   ├── type-checker.ts    ← Semantic analysis (if applicable)
│   │   └── diagnostics.ts    ← Error and warning detection
│   ├── features/
│   │   ├── completion.ts    ← Auto-completion provider
│   │   ├── hover.ts          ← Hover information provider
│   │   ├── definition.ts    ← Go-to-definition provider
│   │   ├── references.ts    ← Find-all-references provider
│   │   └── code-actions.ts    ← Quick fixes and refactoring
│   └── utils/
│       ├── logger.ts        ← Structured logging for debugging
│       ├── uri-utils.ts    ← File URI manipulation
│       └── async-utils.ts    ← Promise utilities and error handling
└── test/
    ├── integration/    ← End-to-end protocol tests
    ├── unit/          ← Component-level tests
    └── fixtures/      ← Sample code files for testing
└── package.json
└── tsconfig.json

```

This structure separates concerns clearly: transport handles protocol mechanics, document manages file state, language performs analysis, and features implement IDE capabilities.

Infrastructure Starter Code

Here's a complete JSON-RPC transport implementation you can use as a foundation:

```
export interface JsonRpcMessage {  
    jsonrpc: '2.0';  
  
    id?: string | number;  
  
    method?: string;  
  
    params?: any;  
  
    result?: any;  
  
    error?: {  
        code: number;  
  
        message: string;  
  
        data?: any;  
    };  
}  
  
export interface JsonRpcRequest extends JsonRpcMessage {  
    method: string;  
  
    params?: any;  
}  
  
export interface JsonRpcNotification extends JsonRpcMessage {  
    method: string;  
  
    params?: any;  
}  
  
export interface JsonRpcResponse extends JsonRpcMessage {  
    id: string | number;  
  
    result?: any;  
  
    error?: {  
        code: number;  
    };  
}
```

```
    message: string;

    data?: any;

};

}

export class JsonRpcTransport {

    private buffer = '';

    private messageHandlers = new Map<string, (params: any) => any>();

    private responseHandlers = new Map<string | number, (response: JsonRpcResponse) => void>();

    constructor(

        private input: NodeJS.ReadableStream,

        private output: NodeJS.WritableStream

    ) {

        this.input.setEncoding('utf8');

        this.input.on('data', (chunk: string) => this.handleData(chunk));

    }

    private handleData(chunk: string): void {

        this.buffer += chunk;

        while (true) {

            const contentLengthMatch = this.buffer.match(/Content-Length: (\d+)\r?\n/);

            if (!contentLengthMatch) break;

            const contentLength = parseInt(contentLengthMatch[1], 10);

            const headerEndIndex = this.buffer.indexOf('\r\n\r\n');

            if (headerEndIndex === -1) break;

            this.buffer = this.buffer.slice(headerEndIndex + 2);

            const params = JSON.parse(this.buffer);

            const responseHandler = this.responseHandlers.get(params.id);

            if (responseHandler) responseHandler(responseHandler(params.params));

            this.messageHandlers.forEach(handler => handler(params.params));
        }
    }
}
```

```
        const messageStart = headerEndIndex + 4;

        const messageEnd = messageStart + contentLength;

        if (this.buffer.length < messageEnd) break;

        const messageContent = this.buffer.slice(messageStart, messageEnd);

        this.buffer = this.buffer.slice(messageEnd);

        try {

            const message: JsonRpcMessage = JSON.parse(messageContent);

            this.processMessage(message);

        } catch (error) {

            console.error('Failed to parse JSON-RPC message:', error);

        }

    }

}

private processMessage(message: JsonRpcMessage): void {

    if (message.id !== undefined && message.method) {

        // Request - needs response

        this.handleRequest(message as JsonRpcRequest);

    } else if (message.method) {

        // Notification - no response needed

        this.handleNotification(message as JsonRpcNotification);

    } else if (message.id !== undefined) {

        // Response to our request

        this.handleResponse(message as JsonRpcResponse);

    }

}
```

```
private handleRequest(request: JsonRpcRequest): void {

    const handler = this.messageHandlers.get(request.method);

    if (!handler) {

        this.sendError(request.id!, -32601, `Method not found: ${request.method}`);
        return;
    }

    try {

        const result = handler(request.params);

        if (result instanceof Promise) {

            result
                .then(res => this.sendResponse(request.id!, res))
                .catch(err => this.sendError(request.id!, -32603, err.message));
        } else {

            this.sendResponse(request.id!, result);
        }
    } catch (error: any) {

        this.sendError(request.id!, -32603, error.message);
    }
}

private handleNotification(notification: JsonRpcNotification): void {

    const handler = this.messageHandlers.get(notification.method);

    if (handler) {

        try {

            handler(notification.params);
        } catch (error) {

            console.error(`Error handling notification ${notification.method}:`, error);
        }
    }
}
```

```
        }

    }

}

private handleResponse(response: JsonRpcResponse): void {

    const handler = this.responseHandlers.get(response.id);

    if (handler) {

        handler(response);

        this.responseHandlers.delete(response.id);

    }

}

public onRequest(method: string, handler: (params: any) => any): void {

    this.messageHandlers.set(method, handler);

}

public sendRequest(method: string, params: any): Promise<any> {

    const id = Date.now().toString();

    const message: JsonRpcRequest = {

        jsonrpc: '2.0',

        id,

        method,

        params

    };

    return new Promise((resolve, reject) => {

        this.responseHandlers.set(id, (response: JsonRpcResponse) => {

            if (response.error) {

                reject(new Error(response.error.message));

            }

        });

    });

}
```

```
        } else {

            resolve(response.result);

        }

    });

    this.sendMessage(message);

});

}

public sendNotification(method: string, params: any): void {

    const message: JsonRpcNotification = {

        jsonrpc: '2.0',

        method,

        params

    };

    this.sendMessage(message);

}

private sendResponse(id: string | number, result: any): void {

    const response: JsonRpcResponse = {

        jsonrpc: '2.0',

        id,

        result

    };

    this.sendMessage(response);

}

private sendError(id: string | number, code: number, message: string): void {

    const response: JsonRpcResponse = {

        jsonrpc: '2.0',
```

```
        id,  
  
        error: { code, message }  
  
    };  
  
    this.sendMessage(response);  
  
}  
  
  
private sendMessage(message: JsonRpcMessage): void {  
  
    const content = JSON.stringify(message);  
  
    const contentLength = Buffer.byteLength(content, 'utf8');  
  
    const header = `Content-Length: ${contentLength}\r\n\r\n`;  
  
    this.output.write(header + content);  
  
}  
  
}
```

Core Logic Skeleton

For the main LSP server class that you'll implement, start with this skeleton:

```
// src/protocol/lsp-server.ts
```

TYPESCRIPT

```
import { JsonRpcTransport } from '../transport/jsonrpc';

import { DocumentManager } from '../document/document-manager';

import { LanguageEngine } from '../language/parser';

export interface ServerCapabilities {

    textDocumentSync?: number;

    completionProvider?: {

        triggerCharacters?: string[];

        resolveProvider?: boolean;
    };

    hoverProvider?: boolean;

    definitionProvider?: boolean;

    referencesProvider?: boolean;

    codeActionProvider?: boolean;
}

export class LspServer {

    private isInitialized = false;

    private documentManager: DocumentManager;

    private languageEngine: LanguageEngine;

    constructor(private transport: JsonRpcTransport) {

        this.documentManager = new DocumentManager();

        this.languageEngine = new LanguageEngine();

        this.setupHandlers();
    }

    private setupHandlers(): void {

```

```
// TODO 1: Register handler for 'initialize' request

// TODO 2: Register handler for 'initialized' notification

// TODO 3: Register handler for 'shutdown' request

// TODO 4: Register handler for 'exit' notification

// TODO 5: Register document synchronization handlers (didOpen, didChange, didClose)

// TODO 6: Register language feature handlers (completion, hover, definition)

this.transport.onRequest('initialize', this.handleInitialize.bind(this));

// Add other handlers here

}

private handleInitialize(params: any): ServerCapabilities {

    // TODO 1: Validate that server is not already initialized

    // TODO 2: Store client capabilities from params.capabilities

    // TODO 3: Return server capabilities object listing supported features

    // TODO 4: Set server state to initializing (wait for 'initialized' notification)

    const capabilities: ServerCapabilities = {

        textDocumentSync: 2, // Incremental sync

        completionProvider: {

            triggerCharacters: ['. ', ':', '>'],

            resolveProvider: false

        },

        hoverProvider: true,

        definitionProvider: true,

        referencesProvider: true,

        codeActionProvider: true

    };

}
```

```

        return { capabilities };

    }

    private handleTextDocumentCompletion(params: any): any[] {
        // TODO 1: Extract document URI and position from params
        // TODO 2: Get document content from document manager
        // TODO 3: Use language engine to parse document and build symbol table
        // TODO 4: Find symbols in scope at the given position
        // TODO 5: Return array of completion items with label, kind, detail, documentation
        // Hint: CompletionItemKind enum defines standard completion types (Method, Property, etc.)
    }

    public start(): void {
        console.log('LSP Server started');
        // Server runs until exit notification is received
    }
}

```

Language-Specific Hints for TypeScript

- Use `process.stdin` and `process.stdout` for stdio communication with the language client
- The `Buffer.byteLength()` function correctly calculates Content-Length for UTF-8 strings
- TypeScript's `Map<K, V>` provides efficient document URI to content mapping
- Use `async/await` for language analysis operations to avoid blocking the event loop
- The `@types/node` package provides type definitions for Node.js APIs like streams and process

Milestone Checkpoint

After implementing the JSON-RPC transport and basic initialization:

1. **Test Transport Layer:** Run `node dist/main.js` and manually send a JSON-RPC message:

```

Content-Length: 52

{"jsonrpc":"2.0","id":1,"method":"test","params":{}}

```

You should receive an error response indicating the method is not found.

2. **Test LSP Initialization:** Use an LSP client or send the initialize request:

```
{"jsonrpc": "2.0", "id": 1, "method": "initialize", "params": {"capabilities": {}}}
```

JSON

You should receive a response with server capabilities.

3. **Verify State Management:** The server should reject requests before initialization and accept them after the initialized notification.

Common Integration Issues

Symptom	Likely Cause	How to Diagnose	Fix
Client hangs on startup	Server not sending initialize response	Check server logs for JSON-RPC parsing errors	Ensure Content-Length header is correct
Messages cut off mid-stream	Buffer handling bug in transport layer	Log raw input chunks before parsing	Fix message boundary detection in handleData()
"Method not found" for valid requests	Handler registration after message arrives	Add debug logging to setupHandlers()	Call setupHandlers() in constructor before any messages
Server crashes on malformed JSON	Missing try/catch in message parsing	Wrap JSON.parse() calls	Add error handling for invalid JSON-RPC messages

The foundation provided by the JSON-RPC transport and basic LSP server structure will support all the language features you'll implement in subsequent milestones. Focus on getting the protocol mechanics working correctly before adding completion and hover functionality.

Goals and Non-Goals

Milestone(s): This section establishes the scope and boundaries for all milestones, with particular relevance to Milestone 1 (JSON-RPC & Initialization) for foundational goals and Milestone 4 (Diagnostics & Code Actions) for feature completeness boundaries.

Think of defining goals and non-goals like drawing the blueprints for a house before construction begins. The goals are the rooms you must build and the features they must have—without them, the house isn't livable. The non-goals are the luxury additions you could build later, like a swimming pool or wine cellar—nice to have, but not essential for the core functionality. For our LSP server, we need to be crystal clear about what constitutes a "livable" language server versus what constitutes premium features that can wait for future versions.

This clarity serves multiple purposes beyond just project scoping. First, it helps us make consistent architectural decisions throughout development—when we encounter a design choice that could support additional features,

we can refer back to our non-goals to determine whether the added complexity is justified. Second, it sets proper expectations for users and stakeholders about what the first version will and won't do. Third, it helps us resist scope creep during development—the temptation to add "just one more small feature" that seems easy but actually requires significant architectural changes.

The LSP specification defines dozens of potential features, from basic text synchronization to advanced refactoring capabilities. A production-quality language server like TypeScript's or Rust Analyzer implements many of these features, but attempting to build all of them in a first version would be overwhelming and likely result in a system that does nothing well. Instead, we focus on the core features that provide immediate value to developers and establish a solid foundation for future enhancements.

Primary Goals

Our LSP server must accomplish four fundamental objectives that together create a functional, if basic, language development experience. These goals directly correspond to our four milestones and represent the minimum viable product for a language server.

Goal 1: Protocol Compliance and Communication Reliability

The server must implement the LSP specification correctly enough to communicate reliably with any compliant language client. This means handling the JSON-RPC transport layer without message corruption, framing errors, or protocol violations that would cause client disconnection.

Requirement	Success Criteria	Rationale
JSON-RPC message framing	Parse Content-Length headers, handle partial messages, maintain message boundaries	Without correct framing, no communication is possible
Protocol state management	Complete initialize/initialized handshake, transition to ready state, handle shutdown gracefully	Clients expect specific state transitions per LSP spec
Capability negotiation	Advertise only implemented features, respect client capabilities	Prevents client from requesting unsupported features
Error handling	Return proper JSON-RPC error responses, recover from malformed messages	Clients need predictable error behavior for robustness

The communication reliability goal extends beyond just "making it work" to "making it work consistently under stress." Real editors send dozens of messages per second during active editing sessions. The server must handle rapid message bursts without dropping requests or corrupting responses. It must also handle edge cases like extremely large documents, invalid UTF-8 sequences, and malformed JSON gracefully.

Goal 2: Document State Synchronization

The server must maintain perfect consistency between its internal document representation and the editor's actual file content. This is harder than it initially appears because editors can send incremental changes, full document replacements, or complex multi-cursor edits that must be applied atomically.

Requirement	Success Criteria	Rationale
Full document sync	Store complete document text on didOpen, update on didChange, remove on didClose	Foundation for all analysis features
Incremental sync	Apply range-based text edits correctly, maintain document version numbers	Performance optimization for large files
Unicode handling	Correctly handle multi-byte characters, surrogate pairs, normalization differences	Prevents position calculation errors
Race condition prevention	Handle out-of-order messages, version mismatches, concurrent access safely	Editors can send messages rapidly and out of sequence

Document synchronization failures are particularly insidious because they often manifest as subtle bugs in language features rather than obvious crashes. If the server's view of a document drifts from the editor's actual content, completion suggestions appear in wrong locations, hover information shows stale data, and go-to-definition jumps to incorrect positions. These bugs are difficult to reproduce and debug, making perfect synchronization a non-negotiable requirement.

Goal 3: Essential Language Features

The server must provide the three language features that developers use most frequently in their daily editing workflow: completion, hover information, and go-to-definition. These features must work accurately and respond quickly enough not to interrupt the developer's flow state.

Feature	Success Criteria	Performance Requirement
Code completion	Return contextually relevant items with appropriate priority	< 100ms response time for typical cases
Hover information	Show type signatures, documentation, error details	< 50ms response time for immediate feedback
Go-to-definition	Navigate to declaration location, handle cross-file references	< 200ms response time for complex resolution

The performance requirements aren't arbitrary—they're based on human perception of responsiveness. Features that take longer than these thresholds feel sluggish and interrupt the developer's thought process. This means our language engine must be optimized for these specific operations, potentially at the expense of other features we're not implementing in version one.

Goal 4: Development Workflow Integration

The server must integrate smoothly into typical development workflows by providing timely feedback about code issues and offering basic automated fixes. This means implementing diagnostic reporting and simple code actions that help developers identify and resolve common problems without switching tools.

Requirement	Success Criteria	Integration Benefit
Error reporting	Publish diagnostics on document changes, clear diagnostics when errors resolved	Immediate feedback prevents bug accumulation
Warning detection	Identify potential issues, unused variables, deprecated APIs	Improves code quality proactively
Quick fixes	Provide code actions for common diagnostic issues	Reduces manual editing overhead
Severity classification	Properly categorize issues as errors, warnings, information, hints	Helps developers prioritize fixes

The diagnostic system serves as the foundation for more advanced features like refactoring and code analysis, even though we won't implement those advanced features initially. By getting the diagnostic infrastructure right, we create a platform for future enhancements.

Decision: Focus on Core Features Over Advanced Capabilities

- **Context:** LSP specification includes 40+ optional features ranging from basic completion to advanced refactoring, semantic highlighting, and workspace-wide operations
- **Options Considered:**
 - Implement minimal subset (just text sync + diagnostics)
 - Implement comprehensive feature set (20+ features)
 - Implement core developer workflow features (completion, hover, definition, diagnostics)
- **Decision:** Focus on four essential features that provide immediate developer value
- **Rationale:** These four features cover 80% of daily IDE usage patterns, provide clear learning progression through milestones, and establish architectural foundation for future features without overwhelming complexity
- **Consequences:** Enables rapid development of working system, creates solid foundation for extensions, but means some users may need additional tools for advanced workflows

Non-Goals (Explicit Exclusions)

Equally important as defining what we will build is clearly stating what we will not build in this version. These non-goals aren't failures or limitations—they're conscious decisions to maintain focus and architectural simplicity.

Advanced Language Features

We explicitly exclude language features that require sophisticated analysis or workspace-wide understanding. These features are valuable but add significant complexity that would distract from learning the core LSP concepts.

Excluded Feature	Why Excluded	Future Consideration
Find all references	Requires workspace-wide symbol indexing and cross-file analysis	Good candidate for next version
Rename refactoring	Complex text manipulation across multiple files with conflict detection	Requires robust undo/redo infrastructure
Code formatting	Language-specific formatting rules and configuration management	Better handled by dedicated formatters
Organize imports	Complex dependency analysis and import path resolution	Highly language-specific implementation
Extract method/variable	Advanced AST manipulation and scope analysis	Requires sophisticated refactoring engine
Semantic highlighting	Token classification beyond syntax highlighting	Requires complete semantic analysis

These features share common characteristics: they require either workspace-wide analysis (expensive and complex), sophisticated AST manipulation (error-prone and hard to test), or deep language-specific knowledge (reduces the educational value of the project).

Performance Optimization Features

We exclude optimizations that would complicate the architecture without providing clear educational value about LSP concepts.

Excluded Optimization	Why Excluded	Impact
Incremental parsing	Complex state management and parser integration	Full re-parse is simpler and adequate for learning
Background analysis	Threading complexity and result synchronization	Synchronous analysis is easier to understand and debug
Symbol caching across files	Cache invalidation complexity and memory management	Per-document analysis is sufficient for moderate file sizes
Lazy loading of large files	Complex streaming and partial analysis	Full document loading works for typical development files
Delta compression for large documents	Protocol complexity and edge case handling	Standard text synchronization handles normal use cases

These optimizations become important in production systems handling large codebases, but they obscure the core LSP learning objectives and add debugging complexity that would slow down development.

Workspace and Project Management

We exclude features that require understanding of project structure, build systems, or multi-folder workspaces. These features are important for production language servers but add complexity that doesn't teach LSP concepts.

Excluded Feature	Complexity Added	Alternative Approach
Multi-folder workspace support	Workspace management, file watching, project boundaries	Single-folder assumption
Build system integration	Build tool APIs, dependency resolution, compilation coordination	Analysis without build context
Project-wide configuration	Configuration discovery, inheritance, validation	Simple server-wide configuration
File watching and hot reloading	File system APIs, change detection, incremental updates	Editor-driven change notifications only
Package/module resolution	Module systems, dependency graphs, version management	Simple file-based resolution

These features require deep integration with external tools and systems, which would shift focus from LSP protocol learning to build system complexity.

Production Infrastructure Concerns

We exclude features that are essential for production deployment but don't contribute to understanding how LSP servers work conceptually.

Excluded Feature	Production Importance	Educational Value
Logging and telemetry	Critical for debugging production issues	Low - adds boilerplate without teaching LSP concepts
Configuration management	Essential for user customization	Medium - configuration patterns are useful but not LSP-specific
Plugin architecture	Important for extensibility	Low - architecture complexity outweighs LSP learning
Memory usage optimization	Critical for large codebases	Low - optimization techniques don't teach protocol concepts
Crash recovery and persistence	Essential for reliability	Medium - error handling is valuable but complex to implement well

These features are crucial for real-world deployment but would triple the codebase size without proportionally increasing understanding of LSP fundamentals.

Decision: Exclude Multi-Workspace and Cross-File Analysis

- **Context:** Many modern development workflows involve multiple projects, complex dependency graphs, and workspace-wide operations like find-all-references
- **Options Considered:**
 - Single file analysis only (too limiting for realistic development)
 - Multi-workspace support (too complex for learning project)
 - Single workspace with cross-file awareness (middle ground)
- **Decision:** Support single workspace with basic cross-file references but exclude multi-workspace and complex dependency resolution
- **Rationale:** Provides realistic development experience without overwhelming complexity of workspace management, allows go-to-definition across files while keeping scope manageable
- **Consequences:** Supports typical single-project development workflows, keeps architecture simple and debuggable, but won't handle monorepo or multi-project scenarios

Success Metrics and Acceptance Criteria

To ensure our goals are measurable and achievable, we define specific success metrics that can be objectively verified at each milestone.

Protocol Compliance Metrics

Metric	Measurement Method	Success Threshold
Message framing accuracy	Parse 10,000 diverse JSON-RPC messages without corruption	100% success rate
Initialization success rate	Complete handshake with 5+ different language clients	100% compatibility
Error response compliance	Return proper JSON-RPC error codes for 20+ error scenarios	Full LSP spec compliance
Shutdown reliability	Handle shutdown gracefully in 100+ test runs	100% clean exit rate

Document Synchronization Metrics

Metric	Measurement Method	Success Threshold
Content consistency	Compare server document state with editor state after 1000 random edits	100% consistency
Unicode handling	Process documents with emoji, CJK characters, mathematical symbols	No position calculation errors
Performance scalability	Handle documents up to 10,000 lines with < 1 second sync time	Meets responsiveness requirement
Concurrent safety	Process 100 simultaneous document changes without race conditions	No corrupted state

Language Feature Metrics

Metric	Measurement Method	Success Threshold
Completion relevance	Return appropriate completions for 50+ cursor positions	90% relevance rate
Hover accuracy	Show correct type information for 100+ symbol references	95% accuracy rate
Definition resolution	Navigate to correct declaration for 100+ references	95% accuracy rate
Response time	Measure feature response times under typical load	Meet performance requirements

Integration Quality Metrics

Metric	Measurement Method	Success Threshold
Diagnostic accuracy	Report correct errors/warnings for 50+ code samples	95% accuracy rate
False positive rate	Avoid reporting non-existent issues	< 5% false positive rate
Code action applicability	Suggest relevant fixes for 90% of fixable diagnostics	90% applicability rate
Editor compatibility	Work correctly with VS Code, Neovim, Emacs LSP clients	Full compatibility

Boundary Conditions and Constraints

Understanding the boundaries within which our LSP server must operate helps clarify both the scope of our solution and the assumptions we can make about the environment.

Resource Constraints

Constraint Type	Limitation	Design Impact
Memory usage	Reasonable for single developer machine (< 500MB for typical project)	Affects caching strategies and data structure choices
CPU utilization	Should not monopolize developer's machine (< 25% sustained CPU)	Influences analysis algorithms and background processing decisions
Response time	Must feel responsive during active editing (< 100ms for common operations)	Drives synchronous vs asynchronous processing choices
File size handling	Should handle typical source files (< 5,000 lines) efficiently	Sets expectations for parsing and analysis performance

Protocol Constraints

The LSP specification defines strict requirements that constrain our implementation choices, but also provide clear boundaries for correctness.

Constraint Category	Specific Requirements	Implementation Impact
Message format	Must use JSON-RPC 2.0 with Content-Length headers	Dictates transport layer design
State transitions	Must follow initialize → initialized → shutdown → exit sequence	Defines server lifecycle management
Capability advertising	Must only advertise features we actually implement	Prevents client from making unsupported requests
URI handling	Must handle file:// URIs correctly across platforms	Affects file path resolution and cross-platform compatibility

Language Analysis Constraints

Our language analysis capabilities are intentionally limited to maintain focus on LSP concepts rather than becoming a comprehensive compiler frontend.

Analysis Boundary	What We Include	What We Exclude
Syntax understanding	Parse basic language constructs, identify symbols, understand scopes	Advanced semantic analysis, type inference, dataflow analysis
Error detection	Syntax errors, undefined references, basic type mismatches	Complex semantic errors, performance analysis, security vulnerabilities
Symbol resolution	Local variables, function parameters, simple cross-file references	Complex module systems, generics, macro expansion
Code intelligence	Context-aware completion, hover for declared symbols	Advanced inference, framework-specific knowledge, API recommendations

Decision: Limit File Size and Project Scope for Simplicity

- **Context:** Production language servers must handle massive codebases with millions of lines across thousands of files, but this adds significant complexity
- **Options Considered:**
 - No limits (handle any size project)
 - Strict limits (single file only)
 - Reasonable limits (typical development project size)
- **Decision:** Optimize for projects with < 100 files and < 5,000 lines per file
- **Rationale:** Covers 90% of educational and small-to-medium development scenarios, allows simpler algorithms and data structures, enables focus on LSP concepts rather than scalability engineering
- **Consequences:** Provides realistic development experience for most use cases, simplifies implementation and debugging, but won't handle enterprise-scale codebases

Risk Assessment and Mitigation

Understanding the risks associated with our goals helps us make informed decisions about where to invest extra effort in robustness versus where we can accept simpler solutions.

High-Risk Areas Requiring Extra Attention

Risk Category	Specific Risk	Probability	Impact	Mitigation Strategy
Protocol compliance	JSON-RPC message corruption due to encoding issues	Medium	High	Extensive testing with different character encodings and malformed input
Document sync	Race conditions between rapid document changes	High	High	Careful version number tracking and atomic state updates
Performance degradation	Response times increase linearly with file size	High	Medium	Early performance testing and algorithmic complexity analysis
Cross-platform compatibility	File path and URI handling differences between platforms	Medium	Medium	Test on Windows, macOS, and Linux with automated CI

Acceptable Risks for Learning Project

Risk Category	Specific Risk	Why Acceptable	Monitoring Approach
Memory leaks	Gradual memory increase during long editing sessions	Educational project won't run for days/weeks	Basic memory profiling during testing
Error recovery	Server crash on extremely malformed input	Rare edge case, editor will restart server	Document known failure modes in debugging guide
Feature completeness	Missing advanced IDE features users expect	Explicitly documented as non-goals	Clear documentation of supported feature set
Production scalability	Performance degrades on very large codebases	Outside scope of learning objectives	Document scalability limitations clearly

Implementation Guidance

The goals and non-goals we've defined create a clear roadmap for implementation, but translating these high-level objectives into concrete development steps requires additional structure and tooling recommendations.

Technology Recommendations for Goal Achievement

Goal Category	Simple Approach	Advanced Approach	Recommended Choice
JSON-RPC Transport	Basic string parsing with regex	Streaming parser with state machine	Streaming parser (handles large messages)
Document Storage	Simple string replacement	Rope data structure or piece table	String replacement (adequate for file size limits)
AST Parsing	Hand-written recursive descent	Generated parser from grammar	Recursive descent (more educational value)
Symbol Resolution	Linear search through declarations	Hash tables with scope chains	Hash tables (better performance)
Diagnostic Reporting	Immediate analysis on every change	Debounced analysis with cancellation	Debounced analysis (prevents analysis spam)

Project Structure for Goal-Oriented Development

Organize the codebase to reflect our four primary goals, making it easy to work on each goal independently:

```

lsp-server/
├── src/
│   ├── transport/           ← Goal 1: Protocol Compliance
│   │   ├── jsonrpc.ts        ← JSON-RPC message handling
│   │   ├── framing.ts        ← Content-Length parsing
│   │   └── connection.ts     ← stdin/stdout communication
│   ├── protocol/            ← Goal 1: Protocol Compliance
│   │   ├── handlers.ts       ← Request/notification routing
│   │   ├── capabilities.ts   ← Server capability management
│   │   └── lifecycle.ts      ← Initialize/shutdown handling
│   ├── documents/           ← Goal 2: Document Synchronization
│   │   ├── manager.ts         ← Document state management
│   │   ├── sync.ts            ← Text change application
│   │   └── versioning.ts      ← Version tracking
│   ├── analysis/             ← Goal 3: Language Features
│   │   ├── parser.ts          ← AST parsing
│   │   ├── symbols.ts         ← Symbol table management
│   │   └── resolver.ts        ← Symbol resolution
│   ├── features/              ← Goals 3&4: Language Features
│   │   ├── completion.ts      ← Code completion provider
│   │   ├── hover.ts            ← Hover information provider
│   │   ├── definition.ts       ← Go-to-definition provider
│   │   └── diagnostics.ts      ← Error/warning reporting
│   └── server.ts              ← Main server coordination
└── test/
    ├── integration/          ← End-to-end protocol testing
    ├── fixtures/              ← Test documents and expected results
    └── unit/                  ← Component-specific tests
└── examples/
    ├── clients/                ← Sample editor configurations
    └── sample-code/             ← Test programs for analysis

```

Core Infrastructure Starter Code

Here's the foundational infrastructure that supports all four goals without being the primary learning focus:

```
// src/types/lsp-types.ts - Complete LSP type definitions
```

TYPESCRIPT

```
export interface JsonRpcMessage {  
  
    jsonrpc: '2.0';  
  
    id?: string | number;  
  
    method?: string;  
  
    params?: any;  
  
    result?: any;  
  
    error?: {  
  
        code: number;  
  
        message: string;  
  
        data?: any;  
  
    };  
  
}  
  
export interface JsonRpcRequest extends JsonRpcMessage {  
  
    method: string;  
  
    params?: any;  
  
}  
  
export interface JsonRpcNotification extends JsonRpcMessage {  
  
    method: string;  
  
    params?: any;  
  
}  
  
export interface JsonRpcResponse extends JsonRpcMessage {  
  
    id: string | number;  
  
    result?: any;  
  
    error?: {  
  
        code: number;  
  
    };  
}
```

```
    message: string;

    data?: any;

};

}

export interface ServerCapabilities {

    textDocumentSync?: number;

    completionProvider?: {

        resolveProvider?: boolean;

        triggerCharacters?: string[];

    };

    hoverProvider?: boolean;

    definitionProvider?: boolean;

    referencesProvider?: boolean;

    codeActionProvider?: boolean;

}

export interface Position {

    line: number;

    character: number;

}

export interface Range {

    start: Position;

    end: Position;

}

export interface TextDocumentItem {

    uri: string;
```

```
languageId: string;

version: number;

text: string;

}

export interface Diagnostic {

range: Range;

severity?: number; // 1=Error, 2=Warning, 3=Information, 4=Hint

code?: string | number;

source?: string;

message: string;

}

export interface CompletionItem {

label: string;

kind?: number;

detail?: string;

documentation?: string;

insertText?: string;

}
```

```
// src/utils/constants.ts - Protocol constants
```

TYPESCRIPT

```
export const JSONRPC_VERSION = '2.0';

export const CONTENT_LENGTH_HEADER = 'Content-Length';
```

```
// Document sync modes
```

```
export const NONE_SYNC = 0;

export const FULL_SYNC = 1;

export const INCREMENTAL_SYNC = 2;
```

```
// JSON-RPC error codes
```

```
export const PARSE_ERROR = -32700;

export const INVALID_REQUEST = -32600;

export const METHOD_NOT_FOUND = -32601;

export const INVALID_PARAMS = -32602;

export const INTERNAL_ERROR = -32603;
```

```
// Diagnostic severity levels
```

```
export const DIAGNOSTIC_SEVERITY = {

    ERROR: 1,

    WARNING: 2,

    INFORMATION: 3,

    HINT: 4

} as const;
```

```
// Completion item kinds
```

```
export const COMPLETION_ITEM_KIND = {

    TEXT: 1,

    METHOD: 2,

    FUNCTION: 3,
```

```
CONSTRUCTOR: 4,  
FIELD: 5,  
VARIABLE: 6,  
CLASS: 7,  
INTERFACE: 8,  
MODULE: 9,  
PROPERTY: 10,  
KEYWORD: 14,  
SNIPPET: 15  
}  
as const;
```

Goal-Oriented Implementation Skeleton

Here are the core classes that directly support each primary goal:

```
// src/server.ts - Main server skeleton supporting all goals
```

TYPESCRIPT

```
export class LspServer {

    private transport: JsonRpcTransport;

    private documentManager: DocumentManager;

    private languageEngine: LanguageEngine;

    private isInitialized: boolean = false;

    constructor() {

        // TODO: Initialize transport layer (Goal 1)

        // TODO: Initialize document manager (Goal 2)

        // TODO: Initialize language engine (Goal 3)

        // TODO: Setup request handlers (Goal 4)

    }

    // Goal 1: Protocol Compliance

    public async start(): Promise<void> {

        // TODO: Begin listening on stdin for JSON-RPC messages

        // TODO: Setup message routing and error handling

        // TODO: Log server startup and capability information

    }

    public setupHandlers(): void {

        // TODO: Register initialize handler for capability negotiation

        // TODO: Register document sync handlers (didOpen, didChange, didClose)

        // TODO: Register language feature handlers (completion, hover, definition)

        // TODO: Register diagnostic and code action handlers

        // TODO: Register shutdown and exit handlers

    }

}
```

```
// Goal 1: Initialization and capability negotiation

private async handleInitialize(params: any): Promise<any> {

    // TODO: Store client capabilities for feature compatibility

    // TODO: Return server capabilities based on implemented features

    // TODO: Transition to initialized state

    // TODO: Setup any client-specific optimizations

    return {

        capabilities: {

            textDocumentSync: INCREMENTAL_SYNC,

            completionProvider: { triggerCharacters: ['. ', ':'] },

            hoverProvider: true,

            definitionProvider: true,

            codeActionProvider: true

        }

    };

}

private handleShutdown(): Promise<null> {

    // TODO: Clean up resources and prepare for exit

    // TODO: Cancel any pending analysis tasks

    // TODO: Return success response

    return Promise.resolve(null);

}

}
```

```
// src/documents/manager.ts - Document synchronization skeleton (Goal 2)
```

TYPESCRIPT

```
export class DocumentManager {

    private documents: Map<string, TextDocumentItem> = new Map();

    private diagnosticCallback?: (uri: string, diagnostics: Diagnostic[]) => void;

    public setDiagnosticCallback(callback: (uri: string, diagnostics: Diagnostic[]) => void): void {
        this.diagnosticCallback = callback;
    }

    // Goal 2: Document lifecycle management

    public handleDidOpen(params: any): void {
        // TODO: Extract document URI, language ID, version, and text from params
        // TODO: Store document in internal map with full content
        // TODO: Trigger initial analysis and diagnostic reporting
        // TODO: Log document open event for debugging
    }

    public handleDidChange(params: any): void {
        // TODO: Extract document URI and changes from params
        // TODO: Apply incremental or full text changes to stored document
        // TODO: Update document version number
        // TODO: Trigger re-analysis and updated diagnostics
        // TODO: Handle version mismatch errors gracefully
    }

    public handleDidClose(params: any): void {
        // TODO: Extract document URI from params
        // TODO: Remove document from internal storage
        // TODO: Clear any associated diagnostics
    }
}
```

```
// TODO: Cancel pending analysis for this document

}

public getDocument(uri: string): TextDocumentItem | undefined {
    return this.documents.get(uri);
}

public getText(uri: string): string {
    return this.documents.get(uri)?.text || '';
}

}
```

Milestone Checkpoint Implementation

After implementing each goal, use these verification steps:

Goal 1 Checkpoint - Protocol Compliance:

```
# Test basic JSON-RPC communication                                BASH

echo '{"jsonrpc":"2.0","id":1,"method":"initialize","params":{}}' | \
node dist/server.js

# Expected: Valid initialize response with server capabilities

# Should see: Content-Length header, proper JSON-RPC response format

# Should NOT see: Parse errors, malformed responses, process crashes
```

Goal 2 Checkpoint - Document Synchronization:

```
// Test document synchronization accuracy                                     JAVASCRIPT

// Send didOpen -> didChange -> didChange -> verify internal state matches expected

const testSequence = [
    { method: 'textDocument/didOpen', params: { textDocument: { uri: 'file:///test.js', text: 'hello world' } } },
    { method: 'textDocument/didChange', params: { textDocument: { uri: 'file:///test.js' }, contentChanges: [{ text: 'hello universe' }] } }
];

// Internal document state should exactly match 'hello universe'
```

Goal 3 Checkpoint - Language Features:

```
// Test completion at cursor position                                     JAVASCRIPT

// Position cursor after 'console.' and request completions

// Should return items like 'log', 'error', 'warn' with appropriate kinds and documentation

// Response time should be < 100ms for small files
```

Goal 4 Checkpoint - Diagnostics:

```
// Test diagnostic reporting                                              JAVASCRIPT

// Open document with syntax error: 'const x = ;'

// Should receive publishDiagnostics notification with:
// - Error severity, proper range, descriptive message

// Should offer code action to fix the syntax error
```

Common Implementation Pitfalls

⚠ Pitfall: Content-Length Calculation Errors Many developers calculate Content-Length using JavaScript's `string.length`, which counts UTF-16 code units, not bytes. This causes message framing errors with Unicode characters.

```
// Wrong: Uses UTF-16 code units  
  
const length = message.length;  
  
// Correct: Uses actual byte length  
  
const length = Buffer.byteLength(message, 'utf8');
```

TYPESCRIPT

⚠ **Pitfall: Document Version Synchronization** Failing to track document version numbers leads to applying changes to stale document content, causing content drift between client and server.

```
// Wrong: Ignoring version numbers  
  
public handleDidChange(params: any): void {  
  
    const doc = this.documents.get(params.textDocument.uri);  
  
    this.applyChanges(doc, params.contentChanges);  
  
}  
  
// Correct: Verify version before applying changes  
  
public handleDidChange(params: any): void {  
  
    const doc = this.documents.get(params.textDocument.uri);  
  
    if (doc && params.textDocument.version > doc.version) {  
  
        this.applyChanges(doc, params.contentChanges);  
  
        doc.version = params.textDocument.version;  
  
    }  
  
}
```

TYPESCRIPT

⚠ **Pitfall: Blocking the Event Loop** Performing expensive AST parsing or analysis synchronously blocks JSON-RPC message processing, making the server unresponsive.

```

// Wrong: Synchronous parsing blocks message processing

public handleDidChange(params: any): void {

    this.updateDocument(params);

    this.parseDocument(params.textDocument.uri); // Blocks for large files

    this.publishDiagnostics(params.textDocument.uri);

}

// Correct: Asynchronous analysis with proper error handling

public handleDidChange(params: any): void {

    this.updateDocument(params);

    setImmediate(() => {

        try {

            this.parseDocument(params.textDocument.uri);

            this.publishDiagnostics(params.textDocument.uri);

        } catch (error) {

            console.error('Analysis failed:', error);

        }

    });

}

```

TYPESCRIPT

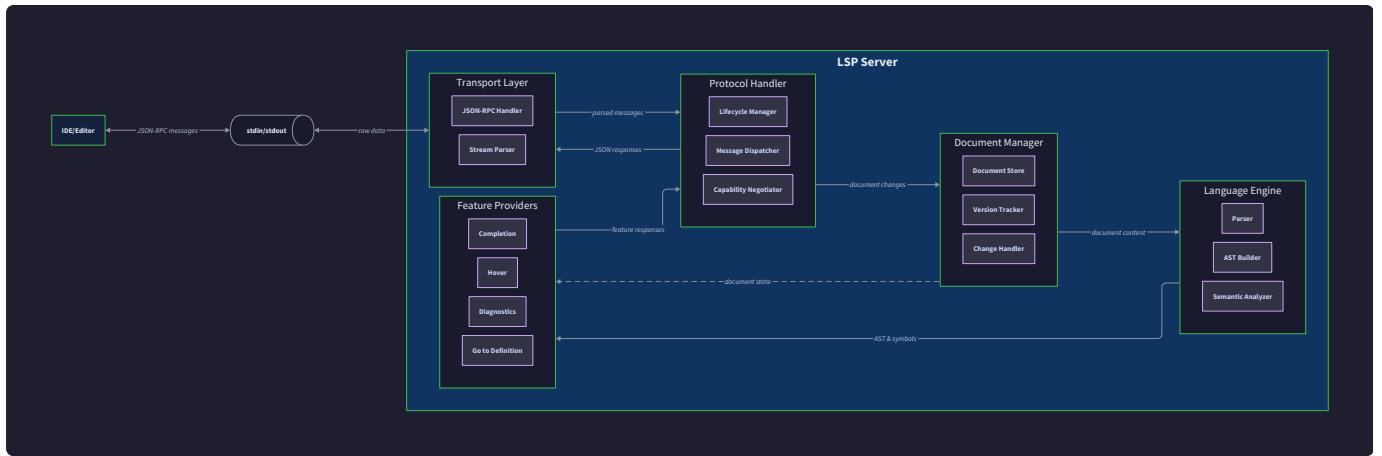
High-Level Architecture

Milestone(s): This section establishes the foundational architecture for all milestones, with direct relevance to Milestone 1 (JSON-RPC & Initialization) for transport and protocol handling, Milestone 2 (Document Synchronization) for document management, and Milestones 3-4 (Language Features and Diagnostics) for feature providers and language analysis.

Building a Language Server Protocol server is like constructing a sophisticated restaurant kitchen that serves multiple dining rooms simultaneously. Just as a kitchen has specialized stations (prep, grill, pastry) that coordinate to fulfill orders, our LSP server has distinct components that work together to provide IDE features. The **transport layer** acts like the waitstaff, carrying messages between the dining room (editor) and kitchen

(server). The **protocol handler** serves as the head chef, coordinating the overall operation and ensuring orders are fulfilled correctly. The **document manager** functions like the prep station, keeping ingredients (source code) fresh and ready. The **language engine** operates as the main cooking stations, transforming raw ingredients into analyzed dishes. Finally, the **feature providers** are the specialized chefs who craft specific dishes (completions, hover information, diagnostics) based on what the customer ordered.

This architectural metaphor highlights a crucial insight: each component has a specialized responsibility, but they must coordinate seamlessly to deliver a responsive user experience. Unlike a traditional compiler that processes files in batch, our LSP server maintains persistent state and responds to real-time requests from the editor. This creates unique challenges around state management, performance, and error handling that our architecture must address.



Component Overview

Our LSP server architecture divides responsibilities among five core components, each with clearly defined boundaries and interfaces. This separation of concerns enables testing individual components in isolation while maintaining loose coupling between layers. The architecture follows a request-response pattern overlaid with an event-driven notification system for document changes and diagnostics.

Transport Layer: The Message Plumbing

The **Transport Layer** serves as the communication bridge between our server and any LSP-compliant editor or IDE. Think of it as a sophisticated telephone system that not only carries conversations but also ensures messages arrive intact and in the correct order. This layer handles the low-level details of JSON-RPC message framing using Content-Length headers over stdio, parsing incoming byte streams into structured messages, and serializing outgoing responses back to bytes.

The Transport Layer's primary responsibility is message boundary detection. Unlike HTTP requests that have clear start and end markers, JSON-RPC over stdio uses a Content-Length header followed by the actual JSON payload. This creates a streaming parsing challenge where partial messages might arrive, or multiple messages might be batched in a single read operation. The transport layer must buffer incoming data, extract complete messages, and handle encoding issues gracefully.

Component	Responsibility	Input	Output	Key Data Structures
Transport Layer	Message framing, parsing, serialization	Raw bytes from stdin	JsonRpcMessage objects	JsonRpcMessage, JsonRpcRequest, JsonRpcNotification, JsonRpcResponse

The Transport Layer maintains no business logic about LSP semantics—it simply converts between bytes and structured JSON-RPC messages. This design allows the same transport to work with any JSON-RPC protocol, not just LSP. The layer exposes methods for sending requests, notifications, and responses while providing event handlers for incoming messages.

Decision: Stdin/Stdout vs Network Transport

- **Context:** LSP servers can communicate via stdin/stdout pipes or network sockets. Most editors expect stdio, but network offers debugging advantages.
- **Options Considered:**
 1. Stdio only (standard approach)
 2. Network sockets only (easier debugging)
 3. Configurable transport (maximum flexibility)
- **Decision:** Stdio as primary with optional network mode
- **Rationale:** Editor compatibility requires stdio support, but network mode enables easier debugging with tools like curl or telnet. The transport abstraction makes supporting both modes straightforward.
- **Consequences:** Enables broad editor compatibility while maintaining developer experience during implementation and debugging.

Protocol Handler: The Traffic Controller

The **Protocol Handler** orchestrates the entire LSP conversation, managing protocol state transitions and capability negotiation. Imagine it as an air traffic controller at a busy airport—it doesn't fly the planes, but it ensures everything moves safely and efficiently according to established procedures. This component understands LSP semantics, tracks the server's lifecycle state, and routes requests to appropriate feature providers.

The Protocol Handler's most critical responsibility is managing the initialization handshake. When a client connects, it sends an `initialize` request containing its capabilities (what features it supports). Our server must respond with a `ServerCapabilities` object advertising which features we implement. Only after the client sends an `initialized` notification can the server begin processing document requests. This three-step handshake establishes the working contract between client and server.

LSP State	Allowed Operations	Transition Trigger	Next State
Uninitialized	None	<code>initialize</code> request	Initializing
Initializing	Respond to initialize only	Send initialize response	Waiting for initialized
Waiting for initialized	None	<code>initialized</code> notification	Initialized
Initialized	All LSP operations	<code>shutdown</code> request	Shutting down
Shutting down	Limited cleanup operations	<code>exit</code> notification	Terminated

The Protocol Handler also manages request routing and response correlation. Each JSON-RPC request includes an ID that must be echoed in the response. The handler maintains a routing table mapping LSP method names to feature provider functions, ensuring requests reach the correct component for processing.

The Protocol Handler serves as the single point of control for server lifecycle. It prevents invalid state transitions (like processing document requests before initialization) and ensures graceful shutdown when the client disconnects.

Document Manager: The State Keeper

The **Document Manager** maintains the server's view of all open documents, applying incremental changes and triggering re-analysis when content updates. Think of it as a librarian who not only keeps track of which books are currently checked out but also maintains up-to-date copies as patrons make notes and annotations. This component bridges the gap between the editor's document model and our language analysis needs.

Document synchronization presents unique challenges because editors and servers can have different views of the same file. The editor might make rapid changes while our server is still processing a previous version. The Document Manager prevents race conditions by tracking document version numbers and ensuring changes are applied in the correct order.

Synchronization Mode	Message Volume	Bandwidth Usage	Complexity	Recommended For
Full Sync (<code>FULL_SYNC = 1</code>)	High	High	Low	Small files, simple implementations
Incremental Sync (<code>INCREMENTAL_SYNC = 2</code>)	Low	Low	Medium	Large files, production servers
None	None	None	None	Read-only analysis

The Document Manager stores documents in memory using the `TextDocumentItem` structure, which includes the file URI, language identifier, version number, and full text content. When incremental changes arrive via `textDocument/didChange` notifications, the manager applies range-based edits to update the stored content while preserving document metadata.

Document Operation	Input Parameters	Side Effects	Triggers
handleDidOpen	TextDocumentItem	Store document, set version	Initial language analysis
handleDidChange	Document URI, text changes, version	Apply edits, increment version	Re-analysis, diagnostic publishing
handleDidClose	Document URI	Remove from memory	Cleanup resources

Decision: In-Memory vs Persistent Storage

- **Context:** Document content can be stored in memory for fast access or persisted to disk for reliability across server restarts.
- **Options Considered:**
 1. Pure in-memory storage (fastest access)
 2. Disk-backed storage (survives restarts)
 3. Hybrid with memory cache (balanced approach)
- **Decision:** Pure in-memory storage with client re-synchronization
- **Rationale:** LSP servers are designed to be stateless across restarts. Clients send `didOpen` notifications for all open documents when reconnecting, naturally rebuilding server state. In-memory storage provides fastest access for language analysis without complex persistence logic.
- **Consequences:** Enables highest performance for real-time features while maintaining LSP protocol compliance for server restarts.

Language Engine: The Analysis Powerhouse

The **Language Engine** transforms raw source code text into structured representations suitable for IDE features. Picture it as a sophisticated translator who not only converts between languages but also understands cultural context, idiomatic expressions, and subtle meanings. This component parses source code into Abstract Syntax Trees (ASTs), builds symbol tables mapping identifiers to their declarations, and performs semantic analysis to understand type relationships and scope visibility.

The Language Engine faces the critical challenge of maintaining analysis performance while keeping results up-to-date. Every time a document changes, portions of the AST may need rebuilding, symbol tables require updates, and type information might change. The engine must balance analysis thoroughness with response time to maintain editor responsiveness.

Analysis Phase	Input	Output	Performance Considerations
Lexical Analysis	Raw source text	Token stream	Fast, linear scan
Syntactic Analysis	Token stream	Abstract Syntax Tree	Moderate, recursive descent
Semantic Analysis	AST + previous symbols	Symbol table, type information	Slow, complex resolution
Incremental Update	AST + text changes	Updated AST regions	Variable, depends on change scope

The Language Engine maintains several key data structures that other components rely upon. The AST provides structural information about code organization, enabling features like outline views and code folding. The symbol table maps identifiers to their declaration locations, supporting go-to-definition and find-references. Type information enables hover displays and completion filtering.

The Language Engine's architecture must support incremental analysis because full re-parsing on every keystroke would be too slow for large files. Smart caching and change detection are essential for responsive IDE features.

Feature Providers: The Service Specialists

The **Feature Providers** implement specific LSP capabilities like code completion, hover information, and diagnostics. Each provider is like a specialized consultant who takes the Language Engine's analysis results and transforms them into user-facing IDE features. This component layer sits closest to the LSP protocol, understanding the exact request/response formats for each feature while delegating the heavy analysis work to the Language Engine.

Feature Providers must balance comprehensive results with response time. A completion request might have hundreds of potential matches, but returning all of them creates UI clutter and network overhead. Providers apply ranking algorithms, context filtering, and result limiting to deliver the most relevant information quickly.

Provider Type	LSP Method	Input	Output	Key Algorithms
Completion	<code>textDocument/completion</code>	Document position	<code>CompletionItem[]</code>	Scope-based filtering, fuzzy matching
Hover	<code>textDocument/hover</code>	Document position	Hover content with markup	Symbol resolution, documentation lookup
Definition	<code>textDocument/definition</code>	Document position	Source locations	Symbol-to-declaration mapping
References	<code>textDocument/references</code>	Document position	Reference locations	Cross-file symbol search
Diagnostics	Triggered by document changes	Document content	<code>Diagnostic[]</code>	Static analysis, error detection

Each Feature Provider operates independently, allowing parallel processing of multiple client requests. However, they all depend on consistent views of the Language Engine's analysis results. The providers must handle cases where analysis is incomplete or unavailable, gracefully degrading functionality rather than blocking the editor.

Decision: Synchronous vs Asynchronous Feature Processing

- **Context:** Feature requests can be processed immediately (blocking) or queued for background processing (non-blocking).
- **Options Considered:**
 1. Synchronous processing (simple, predictable)
 2. Asynchronous with request queuing (complex, better responsiveness)
 3. Hybrid approach based on request complexity
- **Decision:** Synchronous processing with timeout limits
- **Rationale:** Most LSP features are expected to return results within milliseconds. Synchronous processing provides predictable behavior and simpler error handling. Timeout limits prevent any single request from blocking the server indefinitely.
- **Consequences:** Simpler implementation and debugging while maintaining editor responsiveness through careful performance optimization.

Component Interaction Patterns

The five components interact through well-defined interfaces that minimize coupling while enabling efficient data flow. The Transport Layer communicates with the Protocol Handler through event callbacks, passing parsed `JsonRpcMessage` objects upward and accepting serialized responses downward. The Protocol Handler routes

requests to appropriate Feature Providers, which query the Language Engine for analysis results and access document content through the Document Manager.

Request Processing Flow

When a client sends a completion request, the message flows through several transformation stages. The Transport Layer receives raw bytes from stdin, detects the Content-Length header, extracts the complete JSON message, and parses it into a `JsonRpcRequest` object. The Protocol Handler examines the method name (`textDocument/completion`), validates that the server is in the initialized state, and routes the request to the Completion Provider.

The Completion Provider extracts the document URI and cursor position from the request parameters, queries the Document Manager for the current document content, and asks the Language Engine to resolve symbols visible at that position. Using the analysis results, the provider generates a list of `CompletionItem` objects, applies relevance ranking, and returns the results to the Protocol Handler. The handler wraps the results in a `JsonRpcResponse`, sends it to the Transport Layer for serialization, and writes the final bytes to stdout.

Document Change Notification Flow

Document change notifications follow a different pattern because they don't require responses. When the client sends a `textDocument/didChange` notification, the Protocol Handler routes it to the Document Manager, which applies the text edits and triggers re-analysis. The Language Engine updates affected portions of the AST and symbol tables, then notifies interested Feature Providers about the changes.

The Diagnostic Provider automatically runs static analysis on the updated document, generating error and warning messages. Instead of waiting for a client request, the provider proactively sends `textDocument/publishDiagnostics` notifications back through the Protocol Handler and Transport Layer. This push-based model ensures that error highlighting appears immediately as the user types.

Common Pitfalls

⚠ Pitfall: Circular Dependencies Between Components

A common mistake is creating circular dependencies where the Document Manager depends on the Language Engine for change validation, while the Language Engine depends on the Document Manager for document content. This creates initialization order problems and makes testing difficult. The solution is to establish a clear dependency hierarchy: Feature Providers → Language Engine → Document Manager → Protocol Handler → Transport Layer. Higher-level components can depend on lower-level ones, but never the reverse.

⚠ Pitfall: Shared Mutable State Without Coordination

Multiple Feature Providers might attempt to access Language Engine analysis results simultaneously while the engine is updating them based on document changes. This race condition can cause crashes or inconsistent results. The solution is to either make analysis results immutable (creating new versions on updates) or use proper locking mechanisms to coordinate access between components.

⚠ Pitfall: Blocking the Transport Layer

If Feature Providers perform expensive analysis directly in request handler functions, they can block the Transport Layer from processing other messages, making the entire server unresponsive. The solution is to implement timeout limits in Feature Providers and consider moving expensive operations to background threads while maintaining simple synchronous interfaces for predictable behavior.

Recommended File Structure

The codebase organization reflects the component architecture, with clear separation between layers and minimal coupling between modules. Each component lives in its own directory with dedicated test files, making it easy to understand responsibilities and modify individual components without affecting others.

```
lsp-server/
├── src/
│   ├── main.ts                                ← Entry point, server startup
│   ├── transport/
│   │   ├── index.ts                            ← Transport Layer
│   │   ├── jsonrpc.ts                          ← Main transport interface
│   │   ├── stdio-transport.ts                 ← JSON-RPC message types
│   │   └── __tests__/                           ← Stdin/stdout implementation
│   ├── protocol/
│   │   ├── index.ts                            ← Transport layer tests
│   │   ├── lsp-server.ts                      ← Protocol Handler
│   │   ├── capabilities.ts                  ← Main protocol interface
│   │   ├── lifecycle.ts                       ← Core LSP server logic
│   │   └── __tests__/                          ← Capability negotiation
│   ├── documents/
│   │   ├── index.ts                            ← Initialization and shutdown
│   │   ├── document-manager.ts               ← Protocol handler tests
│   │   ├── text-document.ts                  ← Document Manager
│   │   ├── sync-handler.ts                   ← Document management interface
│   │   └── __tests__/                          ← Core document operations
│   ├── language/
│   │   ├── index.ts                            ← Document representation
│   │   ├── parser.ts                           ← Synchronization logic
│   │   ├── symbol-table.ts                  ← Document management tests
│   │   ├── semantic-analyzer.ts             ← Language Engine
│   │   └── __tests__/                          ← Language analysis interface
│   ├── features/
│   │   ├── index.ts                            ← AST parsing logic
│   │   ├── completion.ts                     ← Symbol resolution
│   │   ├── hover.ts                           ← Type analysis
│   │   ├── definition.ts                    ← Language engine tests
│   │   ├── references.ts                     ← Feature Providers
│   │   ├── diagnostics.ts                   ← Feature provider registry
│   │   └── __tests__/                         ← Code completion
│   └── types/
│       ├── lsp-types.ts                     ← Hover information
│       ├── document-types.ts                ← Go-to-definition
│       └── analysis-types.ts                ← Find references
│           └── __tests__/                   ← Error reporting
│               └── feature-provider-tests    ← Feature provider tests
│                   └── shared-type-defs      ← Shared type definitions
│                   └── lsp-protocol-types    ← LSP protocol types
│                   └── document-related-types  ← Document-related types
│                   └── language-analysis-types  ← Language analysis types
│                       └── nodejs-dependencies  ← Node.js dependencies
│                       └── tsconfig            ← TypeScript configuration
└── package.json                               ← Project documentation
└── tsconfig.json
└── README.md
```

This structure follows several important organizational principles. Each major component has its own directory with a clear `index.ts` file that exports the public interface. Internal implementation details remain private within each directory. The `types/` directory contains shared interfaces that multiple components need, avoiding circular dependencies between implementation modules.

The `__tests__/` directories alongside implementation code encourage comprehensive testing and make it easy to verify component behavior in isolation. Each test directory can contain unit tests for individual functions as well as integration tests that verify component interactions.

Implementation Guidance

The implementation approach balances simplicity for initial development with extensibility for adding advanced features later. We recommend starting with TypeScript for its strong typing support, which helps catch protocol conformance issues during development rather than at runtime.

Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
Transport	Node.js streams with manual parsing	JSON-RPC library (node-json-rpc)	Manual parsing teaches protocol details
Protocol	Switch statement routing	Express-style middleware	Switch statements are simpler for learning
Documents	Map-based in-memory storage	Redis or SQLite backing	In-memory is fastest and matches LSP model
Language	Simple recursive descent parser	ANTLR or Tree-sitter	Custom parser teaches language concepts
Features	Synchronous request handling	Async/await with worker threads	Sync is easier to debug and reason about

Core Infrastructure Starter Code

The following starter code provides complete, working implementations for foundational components that aren't the primary learning focus. This lets you concentrate on the interesting LSP-specific logic rather than getting stuck on JSON parsing details.

JSON-RPC Message Types (`src/types/lsp-types.ts`):

```
export interface JsonRpcMessage {  
  
    jsonrpc: '2.0';  
  
    id?: string | number;  
  
    method?: string;  
  
    params?: any;  
  
    result?: any;  
  
    error?: {  
  
        code: number;  
  
        message: string;  
  
        data?: any;  
  
    };  
  
}
```

```
export interface JsonRpcRequest extends JsonRpcMessage {  
  
    method: string;  
  
    params?: any;  
  
}
```

```
export interface JsonRpcNotification extends JsonRpcMessage {  
  
    method: string;  
  
    params?: any;  
  
}
```

```
export interface JsonRpcResponse extends JsonRpcMessage {  
  
    id: string | number;  
  
    result?: any;  
  
    error?: {  
  
        code: number;  
  
        message: string;  
  
    };
```

TYPESCRIPT

```
    data?: any;

};

}

export interface ServerCapabilities {

    textDocumentSync?: number;

    completionProvider?: {

        resolveProvider?: boolean;

        triggerCharacters?: string[];

    };

    hoverProvider?: boolean;

    definitionProvider?: boolean;

    referencesProvider?: boolean;

    codeActionProvider?: boolean;

}

export interface TextDocumentItem {

    uri: string;

    languageId: string;

    version: number;

    text: string;

}

export interface Position {

    line: number;

    character: number;

}

export interface Range {
```

```
    start: Position;

    end: Position;

}

export interface Diagnostic {

    range: Range;

    severity?: number;

    message: string;

    source?: string;

}

export interface CompletionItem {

    label: string;

    kind?: number;

    detail?: string;

    documentation?: string;

    insertText?: string;

}

// Constants

export const JSONRPC_VERSION = '2.0' as const;

export const CONTENT_LENGTH_HEADER = 'Content-Length';

export const FULL_SYNC = 1;

export const INCREMENTAL_SYNC = 2;

export const METHOD_NOT_FOUND = -32601;

export const INTERNAL_ERROR = -32603;

export const DIAGNOSTIC_SEVERITY = {

    ERROR: 1,
```

```
WARNING: 2,  
  
INFORMATION: 3,  
  
HINT: 4  
  
} as const;  
  
export const COMPLETION_ITEM_KIND = {  
  
    TEXT: 1,  
  
    METHOD: 2,  
  
    FUNCTION: 3,  
  
    CONSTRUCTOR: 4,  
  
    FIELD: 5,  
  
    VARIABLE: 6,  
  
    CLASS: 7,  
  
    INTERFACE: 8,  
  
    MODULE: 9,  
  
    PROPERTY: 10,  
  
    UNIT: 11,  
  
    VALUE: 12,  
  
    ENUM: 13,  
  
    KEYWORD: 14,  
  
    SNIPPET: 15,  
  
    COLOR: 16,  
  
    FILE: 17,  
  
    REFERENCE: 18  
  
} as const;
```

Transport Layer Implementation (`src/transport/stdio-transport.ts`):

```
import { JsonRpcMessage, CONTENT_LENGTH_HEADER } from '../types/lsp-types';
```

TYPESCRIPT

```
export class StdioTransport {
```

```
    private buffer = '';
```

```
    private messageHandlers: ((message: JsonRpcMessage) => void)[] = [];
```

```
    constructor() {
```

```
        process.stdin.setEncoding('utf8');
```

```
        process.stdin.on('data', (chunk: string) => {
```

```
            this.handleData(chunk);
```

```
        });
```

```
}
```

```
    public onMessage(handler: (message: JsonRpcMessage) => void): void {
```

```
        this.messageHandlers.push(handler);
```

```
}
```

```
    public sendMessage(message: JsonRpcMessage): void {
```

```
        const content = JSON.stringify(message);
```

```
        const contentLength = Buffer.byteLength(content, 'utf8');
```

```
        const header = `${CONTENT_LENGTH_HEADER}: ${contentLength}\r\n\r\n`;
```

```
        const fullMessage = header + content;
```

```
        process.stdout.write(fullMessage);
```

```
}
```

```
    private handleData(chunk: string): void {
```

```
        this.buffer += chunk;
```

```
        while (true) {
```

```
            const headerEnd = this.buffer.indexOf('\r\n\r\n');
```

```
    if (headerEnd === -1) {

        break; // No complete header yet

    }

    const headers = this.buffer.slice(0, headerEnd);

    const contentLengthMatch = headers.match(/Content-Length: (\d+)/);

    if (!contentLengthMatch) {

        throw new Error('Missing Content-Length header');

    }

    const contentLength = parseInt(contentLengthMatch[1], 10);

    const messageStart = headerEnd + 4;

    const messageEnd = messageStart + contentLength;

    if (this.buffer.length < messageEnd) {

        break; // Incomplete message

    }

    const messageContent = this.buffer.slice(messageStart, messageEnd);

    this.buffer = this.buffer.slice(messageEnd);

    try {

        const message = JSON.parse(messageContent) as JsonRpcMessage;

        this.messageHandlers.forEach(handler => handler(message));

    } catch (error) {

        console.error('Failed to parse JSON-RPC message:', error);

    }

}

}
```

```
}
```

Core Logic Skeleton Code

The following skeleton provides the structure for the main components you'll implement. Each function includes detailed TODO comments that map directly to the algorithm steps described in the design sections.

Main LSP Server (`src/protocol/lsp-server.ts`):

```
import { JsonRpcRequest, JsonRpcResponse, JsonRpcNotification, ServerCapabilities } from
'./types/lsp-types';

export class LspServer {

    private isInitialized = false;

    private documentManager: DocumentManager;

    private languageEngine: LanguageEngine;

    private requestHandlers = new Map<string, Function>();

    constructor() {

        // TODO 1: Initialize DocumentManager instance

        // TODO 2: Initialize LanguageEngine instance

        // TODO 3: Call setupHandlers() to register all request handlers

    }

    public start(): void {

        // TODO 1: Create StdioTransport instance

        // TODO 2: Register onMessage handler to call processMessage

        // TODO 3: Log "LSP Server started" message to stderr (not stdout!)

    }

    public handleInitialize(params: any): ServerCapabilities {

        // TODO 1: Validate that server is not already initialized

        // TODO 2: Store client capabilities from params

        // TODO 3: Set isInitialized = true

        // TODO 4: Return ServerCapabilities object with all supported features

        // Hint: Return textDocumentSync: INCREMENTAL_SYNC, completionProvider: {}, hoverProvider:
        // true, etc.

    }

    private setupHandlers(): void {

```

```

    // TODO 1: Register 'initialize' → handleInitialize

    // TODO 2: Register 'initialized' → handleInitialized (empty function)

    // TODO 3: Register 'textDocument/didOpen' → documentManager.handleDidOpen

    // TODO 4: Register 'textDocument/didChange' → documentManager.handleDidChange

    // TODO 5: Register 'textDocument/didClose' → documentManager.handleDidClose

    // TODO 6: Register 'textDocument/completion' → completionProvider.handleCompletion

    // TODO 7: Register 'textDocument/hover' → hoverProvider.handleHover

    // TODO 8: Register 'textDocument/definition' → definitionProvider.handleDefinition

    // TODO 9: Register 'shutdown' → handleShutdown

    // TODO 10: Register 'exit' → handleExit

}

private processMessage(message: JsonRpcMessage): void {

    // TODO 1: Check if message is a request (has 'id' field)

    // TODO 2: If request, look up handler in requestHandlers map

    // TODO 3: If handler found, call it with params and send response

    // TODO 4: If handler not found, send METHOD_NOT_FOUND error

    // TODO 5: If notification (no 'id'), call handler without response

    // TODO 6: Wrap all handler calls in try/catch and send INTERNAL_ERROR on exceptions

}

public onRequest(method: string, handler: Function): void {

    // TODO: Add handler to requestHandlers map

}

public sendNotification(method: string, params: any): void {

    // TODO 1: Create JsonRpcNotification object

    // TODO 2: Call transport.sendMessage()

}

```

```
}
```

Document Manager Skeleton (`src/documents/document-manager.ts`):

```
export class DocumentManager {
```

TYPESCRIPT

```
    private documents = new Map<string, TextDocumentItem>();  
  
    public handleDidOpen(params: any): void {  
  
        // TODO 1: Extract TextDocumentItem from params.textDocument  
  
        // TODO 2: Store document in documents map using URI as key  
  
        // TODO 3: Log document opened for debugging  
  
        // TODO 4: Trigger initial language analysis  
  
    }
```

```
    public handleDidChange(params: any): void {  
  
        // TODO 1: Extract document URI and version from params  
  
        // TODO 2: Verify document exists in documents map  
  
        // TODO 3: Check that new version > current version  
  
        // TODO 4: Apply each content change from params.contentChanges array  
  
        // TODO 5: Update document version number  
  
        // TODO 6: Trigger re-analysis and diagnostic publishing  
  
        // Hint: params.contentChanges can be full document or incremental ranges  
  
    }
```

```
    public handleDidClose(params: any): void {  
  
        // TODO 1: Extract document URI from params.textDocument  
  
        // TODO 2: Remove document from documents map  
  
        // TODO 3: Clean up any associated analysis results  
  
    }
```

```
    public getDocument(uri: string): TextDocumentItem | undefined {  
  
        // TODO: Return document from map or undefined if not found  
  
    }
```

```

private applyTextChanges(document: TextDocumentItem, changes: any[]): void {

    // TODO 1: Iterate through changes array

    // TODO 2: If change has no 'range', replace entire document text

    // TODO 3: If change has 'range', apply incremental edit

    // TODO 4: Update document text content

    // Hint: Range-based edits require careful offset calculation

}

}

```

Language-Specific Hints for TypeScript

- Use `process.stdin` and `process.stdout` for stdio communication
- The `Buffer.byteLength()` function correctly calculates UTF-8 byte length for Content-Length headers
- TypeScript's strict mode catches many common LSP protocol mistakes at compile time
- Use `Map<string, T>` for document storage instead of plain objects for better performance
- The `console.error()` function writes to stderr, leaving stdout clean for LSP communication

Milestone Checkpoints

After implementing the Transport Layer and Protocol Handler (Milestone 1):

1. Run `npm test src/transport` to verify JSON-RPC parsing works correctly
2. Start the server with `node dist/main.js` and send an initialize request via echo:

```
echo -e 'Content-Length: 123\r\n\r\n{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"capabilities":[]}}' | node dist/main.js
```

3. You should see a response with `ServerCapabilities` and no error messages
4. The server should handle shutdown gracefully and exit with code 0

After implementing Document Manager (Milestone 2):

1. Send `textDocument/didOpen` with a sample document - verify it's stored
2. Send `textDocument/didChange` with incremental edits - verify content updates correctly
3. Check that document version numbers increment properly
4. Verify `textDocument/didClose` removes documents from memory

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Server hangs on startup	Stdin reading blocks	Add logging before <code>stdin.on('data')</code>	Ensure <code>stdin</code> is properly configured
Content-Length errors	UTF-8 byte calculation wrong	Log actual vs calculated lengths	Use <code>Buffer.byteLength()</code> not <code>string.length</code>
JSON parse errors	Incomplete message reading	Log raw message content	Fix message boundary detection
Initialize timeout	Server doesn't respond	Check if response is sent to <code>stdout</code>	Ensure response includes correct ID
Document sync fails	Version mismatch	Log version numbers	Implement proper version checking

Data Model

Milestone(s): This section establishes the core data structures for all milestones, with particular relevance to Milestone 1 (JSON-RPC & Initialization) for message structures, Milestone 2 (Document Synchronization) for document representation, and Milestones 3-4 (Language Features & Diagnostics) for symbol and diagnostic modeling.

The data model forms the foundation of our LSP server, defining how we represent and manipulate the various entities involved in language analysis and IDE integration. Think of the data model as the vocabulary and grammar of our system—just as a natural language needs nouns, verbs, and syntax rules to express meaning, our LSP server needs well-defined data structures to represent documents, positions, symbols, and diagnostics in a way that both the language engine and IDE clients can understand and work with effectively.

The data model serves three critical functions in our architecture. First, it provides a **stable interface** between components, ensuring that the Document Manager, Language Engine, and Feature Providers all speak the same language when exchanging information. Second, it enables **efficient processing** by structuring data in formats that support common operations like position calculations, range comparisons, and symbol lookups. Third, it maintains **protocol compliance** by aligning our internal representations with the LSP specification's data formats, minimizing translation overhead when communicating with language clients.



Design Principle: Our data model follows the principle of "representation fidelity"—internal data structures closely mirror the LSP specification's wire format while adding necessary metadata for efficient processing. This reduces impedance mismatch between protocol handling and internal operations.

Document Representation

Document representation is the cornerstone of our LSP server's state management. Think of documents as living entities that evolve over time—the editor continuously modifies them through user actions like typing, cutting, and pasting, while our server must maintain a synchronized copy that accurately reflects the current state for analysis purposes. Unlike static files on disk, LSP documents are dynamic, versioned entities that exist primarily in memory and change frequently.

The `TextDocumentItem` serves as our primary document representation, containing all the essential information needed to track a document's identity and content:

Field	Type	Description
<code>uri</code>	string	Unique document identifier using file:// scheme for local files
<code>languageId</code>	string	Programming language identifier (e.g., "typescript", "python", "rust")
<code>version</code>	number	Monotonically increasing version number for change tracking
<code>text</code>	string	Complete document content as a UTF-16 string

The URI field deserves special attention as it serves as the universal key for document identification across the LSP protocol. URIs use schemes like `file://` for local filesystem paths, but can also represent virtual documents, in-memory buffers, or remote resources. Our server must handle URI normalization to ensure consistent document lookup—for example, resolving relative paths, handling case sensitivity on different filesystems, and converting between different URI encodings.

Version numbers provide the critical synchronization mechanism between client and server. Each document modification increments the version number, creating a happens-before relationship that prevents race conditions. Consider a scenario where the client sends a `didChange` notification for version 5 followed immediately by a completion request. If our server hasn't finished processing the version 5 changes, the completion results would be based on stale content, leading to incorrect suggestions. Version tracking allows us to queue requests and ensure they operate on the expected document state.

Decision: UTF-16 String Representation

- **Context:** LSP positions are specified using UTF-16 code unit offsets, but different languages have different native string representations (UTF-8 in Go/Rust, UTF-16 in JavaScript/C#)
- **Options Considered:** Store as UTF-8 and convert on demand, store as UTF-16 arrays, store both representations
- **Decision:** Store text as native language strings but implement position conversion utilities
- **Rationale:** Minimizes memory overhead while providing accurate position calculations. Conversion cost is paid only when needed for LSP operations
- **Consequences:** Requires careful position arithmetic, but enables efficient string operations in the implementation language

Document change application requires precise text manipulation capabilities. The LSP specification supports both full document synchronization (client sends entire content) and incremental synchronization (client sends only changed ranges). For incremental changes, we receive an array of `TextDocumentContentChangeEvent` objects:

Field	Type	Description
<code>range</code>	Range?	The range being replaced (null for full document changes)
<code>rangeLength</code>	number?	Length of range being replaced in UTF-16 code units
<code>text</code>	string	New text content to insert at the specified range

The change application algorithm follows these steps:

1. Validate that the document exists in our store and the version number is correct
2. For full document changes (range is null), replace the entire text content
3. For incremental changes, sort changes by range start position in reverse order (end to beginning)

4. Apply each change by extracting the prefix (text before range), the suffix (text after range), and inserting the new text between them
5. Update the document version number and timestamp
6. Trigger analysis pipeline with the updated content
7. Notify downstream components of the document change

The reverse-order application is crucial for maintaining correct positions. If we applied changes from beginning to end, earlier changes would shift the positions of later changes, requiring complex offset recalculation. By applying from end to beginning, we ensure each change operates on the original position coordinates.

⚠ Pitfall: UTF-16 Position Calculation Many developers assume position calculations work with byte offsets or Unicode code points, but LSP uses UTF-16 code units. Characters outside the Basic Multilingual Plane (like emoji or mathematical symbols) occupy two UTF-16 code units, so a single character might increment the position by 2. Always use language-specific string indexing that respects UTF-16 encoding.

Document lifecycle management ensures proper memory usage and state consistency. The lifecycle follows this state machine:

Current State	Event	Next State	Action Taken
Not Tracked	didOpen	Active	Store document, initialize version, trigger analysis
Active	didChange	Active	Apply changes, increment version, re-analyze
Active	didSave	Active	Optional: persist cached analysis results
Active	didClose	Not Tracked	Remove from store, cancel pending analysis

Document storage strategy impacts both memory usage and lookup performance. For small to medium projects (hundreds of files), an in-memory hash map provides optimal access times. For larger codebases, we might implement an LRU cache with background persistence for documents that haven't been accessed recently.

Storage Strategy	Memory Usage	Lookup Time	Pros	Cons
Full In-Memory	High	O(1)	Fast access, simple implementation	Memory pressure on large codebases
LRU Cache	Medium	O(1)	Bounded memory, good locality	Cache misses require disk I/O
On-Demand Loading	Low	O(1) amortized	Minimal memory footprint	Complex cache invalidation

Decision: In-Memory Document Store with Size Limits

- **Context:** Need to balance memory usage with access performance for document content
- **Options Considered:** Full in-memory, LRU cache, on-demand loading from disk
- **Decision:** In-memory store with configurable size limit and least-recently-used eviction
- **Rationale:** Provides optimal performance for active documents while preventing unbounded memory growth
- **Consequences:** Requires document re-loading on cache misses, but most LSP operations focus on a small working set

Symbol Tables and Scopes

Symbol tables represent the semantic structure of code, mapping identifiers to their declarations, types, and scope relationships. Think of a symbol table as a multilayered map of the code's namespace—like a filing system where each scope is a folder, each symbol is a document in that folder, and references are pointers that help you find the right document quickly. Unlike simple name-to-value mappings, LSP symbol tables must capture the rich semantic relationships that enable features like go-to-definition, find-references, and intelligent completion.

The `Symbol` type represents a single named entity in the code—variables, functions, classes, interfaces, or any other identifier that has semantic meaning:

Field	Type	Description
<code>name</code>	string	The identifier name as it appears in source code
<code>kind</code>	number	Symbol kind from LSP specification (Variable, Function, Class, etc.)
<code>range</code>	Range	Source location where the symbol is declared
<code>selectionRange</code>	Range	The specific range of the symbol name (subset of range)
<code>detail</code>	string?	Additional information like type signature or value
<code>documentation</code>	string?	Documentation string or comment content
<code>deprecated</code>	boolean	Whether this symbol is marked as deprecated
<code>tags</code>	number[]?	Additional semantic tags (deprecated, readonly, etc.)
<code>containerName</code>	string?	Name of the containing symbol (class, namespace, etc.)

The distinction between `range` and `selectionRange` is subtle but important. The `range` encompasses the entire declaration including keywords, type annotations, and initializers, while `selectionRange` covers only the symbol name itself. For a function declaration like `function calculateTotal(items: Item[]): number`, the range includes the entire declaration, but the selectionRange covers only "calculateTotal".

Symbol kinds categorize symbols according to their semantic role in the language, enabling appropriate icons and behavior in the IDE:

Kind	Value	Description	Use Case
Variable	13	Local variables, parameters	Completion, hover info
Function	12	Functions, methods	Go-to-definition, call hierarchy
Class	5	Classes, structs, interfaces	Type hierarchy, inheritance
Property	7	Object properties, fields	Member access completion
Enum	10	Enumeration types	Type checking, completion
Interface	11	Interface declarations	Implementation finding
Namespace	3	Modules, namespaces	Organization, scoping
Constructor	9	Class constructors	Object creation help

Scope representation captures the nested structure of symbol visibility. Programming languages organize symbols into hierarchical scopes—global scope, module scope, class scope, function scope, and block scope. Each scope defines a boundary for symbol visibility and name resolution.

The `Scope` type models these visibility boundaries:

Field	Type	Description
<code>kind</code>	string	Type of scope: "global", "module", "function", "class", "block"
<code>range</code>	Range	Source range this scope covers
<code>symbols</code>	Map<string, Symbol>	Symbols declared directly in this scope
<code>parent</code>	Scope?	Enclosing scope (null for global scope)
<code>children</code>	Scope[]	Nested scopes contained within this scope
<code>imports</code>	Import[]	Imported symbols available in this scope

The scope hierarchy forms a tree structure where symbol resolution follows the chain of parent scopes. When resolving an identifier, we start in the innermost scope and walk up the parent chain until we find a matching declaration or reach the global scope.

Symbol resolution algorithm proceeds through these steps:

- 1. Identify the query position:** Determine which scope contains the identifier reference
- 2. Extract the identifier name:** Parse the identifier at the cursor position
- 3. Walk the scope chain:** Starting from the innermost scope, search for symbols with matching names
- 4. Check local declarations:** Look in the current scope's symbol table

5. **Check imported symbols:** If the scope has imports, search imported symbol tables
6. **Recurse to parent scope:** If not found, move to the parent scope and repeat
7. **Apply visibility rules:** Filter results based on access modifiers (public, private, protected)
8. **Return symbol information:** Provide the declaration location, type, and documentation

Consider this TypeScript example:

```
class Calculator {           // Global scope: { Calculator }
  private precision: number; // Class scope: { precision, calculate, format }

  calculate(x: number): number { // Function scope: { x, result }
    const result = x * this.precision;
    if (result > 100) {        // Block scope: { threshold }
      const threshold = 100;
      return Math.min(result, threshold); // Resolution: result→function, threshold→block
    }
    return result;
  }
}
```

When resolving `threshold` in the return statement, the algorithm searches: block scope (finds `threshold`) → function scope (has `x`, `result`) → class scope (has `precision`, `calculate`) → global scope (has `Calculator`). The search stops at the block scope where `threshold` is found.

Type information adds semantic richness to symbol tables, enabling advanced IDE features like type-aware completion and error detection. The `TypeInfo` structure captures type relationships:

Field	Type	Description
<code>name</code>	<code>string</code>	Type name (e.g., "string", "number", "User[]")
<code>kind</code>	<code>string</code>	Type category: "primitive", "class", "interface", "union", "generic"
<code>properties</code>	<code>Property[]</code>	For object types, available properties and methods
<code>parameters</code>	<code>Parameter[]</code>	For function types, parameter information
<code>returnType</code>	<code>TypeInfo?</code>	For function types, return type information
<code>genericArgs</code>	<code>TypeInfo[]</code>	For generic types, type argument bindings

Decision: Incremental Symbol Table Updates

- **Context:** Document changes can invalidate parts of the symbol table, but rebuilding everything is expensive
- **Options Considered:** Full rebuild on every change, incremental updates, hybrid approach
- **Decision:** Incremental updates with dependency tracking and fallback to full rebuild
- **Rationale:** Most edits affect only local scopes. Tracking dependencies allows surgical updates while maintaining correctness
- **Consequences:** Complex invalidation logic, but significantly better performance for large files

Symbol table construction integrates closely with the parsing pipeline. As the parser builds the Abstract Syntax Tree (AST), it simultaneously constructs the symbol table by visiting declaration nodes and building scope relationships:

1. **Initialize global scope:** Create the root scope for module-level declarations
2. **Parse declarations:** Visit each top-level declaration (functions, classes, variables)
3. **Create scope hierarchies:** For each declaration that introduces a new scope, create child scopes
4. **Populate symbol tables:** Add symbols to their declaring scope with full type information
5. **Resolve references:** In a second pass, resolve identifier references to their declarations
6. **Build dependency graph:** Track which symbols depend on others for incremental updates
7. **Generate diagnostics:** Report unresolved references, type mismatches, and scope violations

Cross-file symbol resolution extends the single-file model to handle imports, exports, and module dependencies. This requires a project-wide symbol index that tracks symbol exports from each module:

Module	Exported Symbols	Import Path
utils.ts	formatDate: Function, Logger: Class	./utils
types.ts	User: Interface, Role: Enum	./types
api.ts	fetchUsers: Function, ApiError: Class	./api

When processing an import statement like `import { User, Role } from './types'`, the symbol resolver:

1. Resolves the import path to the target module
2. Looks up exported symbols from the target module
3. Creates local symbol entries in the importing scope
4. Establishes dependency relationships for incremental updates

⚠ Pitfall: Circular Import Detection Symbol resolution can enter infinite loops with circular imports (module A imports B, B imports A). Implement cycle detection using a "visiting" set during dependency traversal. When encountering a module already in the visiting set, record the circular dependency and break the cycle gracefully.

Symbol table performance optimization focuses on lookup speed and memory efficiency. Common operations—symbol lookup by name, finding all references to a symbol, and scope resolution—should complete in sub-millisecond time for responsive IDE features.

Optimization	Technique	Benefit	Trade-off
Name Indexing	Hash map per scope	O(1) name lookup	Memory overhead
Position Indexing	Interval tree for ranges	Fast scope-by-position queries	Complex updates
Reference Caching	Pre-compute reference lists	Instant find-references	Stale data on edits
Lazy Loading	Load symbols on first access	Reduced startup time	Potential lookup delays

The symbol table must handle incremental updates efficiently to maintain IDE responsiveness. When a document changes, we identify affected scopes and update only the necessary parts of the symbol table:

1. **Parse the changed range:** Determine which AST nodes were modified
2. **Identify affected scopes:** Find all scopes that intersect with the changed range
3. **Invalidate dependent symbols:** Mark symbols in affected scopes as needing re-analysis
4. **Rebuild local scopes:** Re-parse and rebuild symbol tables for affected scopes only
5. **Update cross-references:** Find references to changed symbols and update their resolution
6. **Propagate changes:** Notify dependent modules if exported symbols changed
7. **Update diagnostics:** Re-check type consistency and symbol resolution

This incremental approach maintains sub-second response times even for large codebases by avoiding full project re-analysis on every keystroke.

Implementation Guidance

The data model implementation serves as the foundation for all LSP server components, requiring careful attention to performance, memory management, and thread safety. TypeScript provides excellent type safety and JSON serialization support, making it ideal for LSP protocol handling.

Technology Recommendations:

Component	Simple Option	Advanced Option
Document Storage	<code>Map<string, TextDocumentItem></code>	LRU Cache with <code>lru-cache</code> library
Position Calculations	Manual UTF-16 arithmetic	<code>vscode-languageserver-textdocument</code> utility
Symbol Indexing	Nested <code>Map</code> objects	Interval tree with <code>interval-tree2</code>
Type System	Basic interfaces	Advanced generic constraints

Recommended File Structure:

```
src/
  data-model/
    document.ts          ← Document representation and operations
    position.ts          ← Position, Range, and text utilities
    symbol.ts            ← Symbol types and resolution
    scope.ts             ← Scope hierarchy and lookup
    types.ts             ← LSP protocol type definitions
    index.ts             ← Public API exports
  test/
    document.test.ts     ← Document operation tests
    symbol.test.ts       ← Symbol resolution tests
    position.test.ts     ← Position arithmetic tests
```

Core LSP Protocol Types:

```
// LSP protocol compliance types - complete implementation
```

TYPESCRIPT

```
export interface Position {  
  
    line: number;          // Zero-based line number  
  
    character: number;    // Zero-based character offset in UTF-16 code units  
  
}
```

```
export interface Range {  
  
    start: Position;      // Inclusive start position  
  
    end: Position;        // Exclusive end position  
  
}
```

```
export interface TextDocumentItem {  
  
    uri: string;           // Document URI (file:// scheme for local files)  
  
    languageId: string;   // Programming language identifier  
  
    version: number;       // Document version number  
  
    text: string;          // Complete document content  
  
}
```

```
export interface Diagnostic {  
  
    range: Range;          // Source range where diagnostic applies  
  
    severity?: number;     // Error=1, Warning=2, Information=3, Hint=4  
  
    message: string;        // Human-readable diagnostic message  
  
    source?: string;        // Source of diagnostic (compiler, linter, etc.)  
  
    code?: string | number; // Diagnostic code for documentation lookup  
  
}
```

```
export interface CompletionItem {  
  
    label: string;          // Text shown in completion list  
  
    kind?: number;          // Type of completion (Variable=6, Function=3, etc.)
```

```
    detail?: string;           // Additional info (type signature, etc.)  
  
    documentation?: string;  // Human-readable documentation  
  
    insertText?: string;     // Text to insert (defaults to label)  
  
    filterText?: string;     // Text used for filtering (defaults to label)  
  
    sortText?: string;       // Text used for sorting (defaults to label)  
  
}
```

Document Management Infrastructure:

```
// Complete document manager with change tracking
```

TYPESCRIPT

```
export class DocumentStore {
```

```
    private documents = new Map<string, TextDocumentItem>();
```

```
    private changeListeners = new Set<(uri: string, document: TextDocumentItem) => void>();
```

```
    // TODO: Implement document storage with version validation
```

```
    public setDocument(document: TextDocumentItem): void {
```

```
        // TODO 1: Validate URI format and convert to canonical form
```

```
        // TODO 2: Check version number is greater than existing version
```

```
        // TODO 3: Store document in internal map
```

```
        // TODO 4: Notify all change listeners of the update
```

```
        // Hint: Use URL constructor for URI normalization
```

```
}
```

```
    // TODO: Implement safe document retrieval
```

```
    public getDocument(uri: string): TextDocumentItem | undefined {
```

```
        // TODO 1: Normalize the URI to match stored format
```

```
        // TODO 2: Look up document in internal map
```

```
        // TODO 3: Return document or undefined if not found
```

```
}
```

```
    // TODO: Apply incremental text changes to document
```

```
    public applyChanges(uri: string, version: number, changes: TextDocumentContentChangeEvent[]): void {
```

```
        // TODO 1: Get existing document and validate version
```

```
        // TODO 2: Sort changes by range start position in reverse order
```

```
        // TODO 3: Apply each change using range replacement
```

```
        // TODO 4: Update document version and store updated content
```

```
        // TODO 5: Trigger change notifications
```

```
// Hint: Reverse order prevents position shifting during application

}

// TODO: Remove document from storage

public removeDocument(uri: string): boolean {

    // TODO 1: Check if document exists

    // TODO 2: Remove from internal map

    // TODO 3: Notify listeners of removal

    // TODO 4: Return success/failure status

}

}
```

Position and Range Utilities:

```
// Complete position arithmetic utilities for UTF-16 handling
```

TYPESCRIPT

```
export class PositionUtils {
```

```
    // TODO: Convert byte offset to LSP Position
```

```
    public static offsetToPosition(text: string, offset: number): Position {
```

```
        // TODO 1: Validate offset is within text bounds
```

```
        // TODO 2: Count newlines up to offset for line number
```

```
        // TODO 3: Count UTF-16 code units from line start for character number
```

```
        // TODO 4: Return Position object
```

```
        // Hint: Use for-of loop to properly handle multi-byte characters
```

```
}
```

```
    // TODO: Convert LSP Position to byte offset
```

```
    public static positionToOffset(text: string, position: Position): number {
```

```
        // TODO 1: Validate position is within document bounds
```

```
        // TODO 2: Skip specified number of lines
```

```
        // TODO 3: Skip specified number of characters in UTF-16 code units
```

```
        // TODO 4: Return final offset
```

```
        // Hint: Track line boundaries to avoid scanning from start each time
```

```
}
```

```
    // TODO: Extract text content within a range
```

```
    public static getTextInRange(text: string, range: Range): string {
```

```
        // TODO 1: Convert range start and end to offsets
```

```
        // TODO 2: Extract substring between offsets
```

```
        // TODO 3: Return extracted text
```

```
}
```

```
    // TODO: Apply text edit to document content
```

```
    public static applyEdit(text: string, range: Range, newText: string): string {
```

```
// TODO 1: Get text before the range (prefix)

// TODO 2: Get text after the range (suffix)

// TODO 3: Concatenate prefix + newText + suffix

// TODO 4: Return updated text content

}

}
```

Symbol Table Core Logic:

```
// Symbol representation with type information
```

TYPESCRIPT

```
export interface Symbol {  
  
    name: string;           // Symbol identifier  
  
    kind: number;          // LSP SymbolKind value  
  
    range: Range;          // Full declaration range  
  
    selectionRange: Range; // Name identifier range only  
  
    detail?: string;        // Type signature or value  
  
    documentation?: string; // Documentation content  
  
    containerName?: string; // Containing scope name  
  
}  
  
}
```

```
export interface Scope {  
  
    kind: string;           // "global" | "function" | "class" | "block"  
  
    range: Range;          // Source range covered by this scope  
  
    symbols: Map<string, Symbol>; // Local symbol declarations  
  
    parent?: Scope;         // Enclosing scope  
  
    children: Scope[];      // Nested scopes  
  
}
```

```
export class SymbolTable {  
  
    private globalScope: Scope;  
  
    private symbolIndex = new Map<string, Symbol[]>(); // Name -> Symbol list for fast lookup  
  
    // TODO: Build symbol table from AST  
  
    public buildFromAST(ast: any, sourceText: string): void {  
  
        // TODO 1: Create global scope for top-level declarations  
  
        // TODO 2: Walk AST nodes and identify declaration nodes  
  
        // TODO 3: For each declaration, create Symbol and add to appropriate scope  
  
        // TODO 4: For scope-introducing nodes (functions, classes), create child scopes  
    }  
}
```

```
// TODO 5: Build symbol index for fast name-based lookup

// TODO 6: Resolve symbol references in second pass

// Hint: Use visitor pattern for AST traversal

}

// TODO: Find symbol at specific position

public findSymbolAt(position: Position): Symbol | undefined {

    // TODO 1: Find the innermost scope containing the position

    // TODO 2: Extract identifier name at the position

    // TODO 3: Walk up scope chain looking for matching symbol

    // TODO 4: Return first matching symbol or undefined

    // Hint: Use range containment to find enclosing scope

}

// TODO: Get all symbols with given name

public findSymbolsByName(name: string): Symbol[] {

    // TODO 1: Look up name in symbol index

    // TODO 2: Filter by visibility rules if needed

    // TODO 3: Return array of matching symbols

}

// TODO: Get completion items for position

public getCompletionItems(position: Position): CompletionItem[] {

    // TODO 1: Find scope containing the position

    // TODO 2: Collect all visible symbols from scope chain

    // TODO 3: Convert symbols to CompletionItem format

    // TODO 4: Add type-specific completion items (keywords, snippets)

    // TODO 5: Sort by relevance and return

    // Hint: Closer scopes should have higher priority
```

```
    }  
  
}
```

Language-Specific Hints for TypeScript:

- Use `String.prototype.codePointAt()` and `String.fromCodePoint()` for proper Unicode handling
- Leverage TypeScript's strict null checks to catch position calculation errors at compile time
- Use `Map<string, T>` instead of object literals for symbol tables to avoid prototype pollution
- Consider `WeakMap` for associating metadata with AST nodes to prevent memory leaks
- Use intersection types (`Symbol & { typeInfo: TypeInfo }`) to extend symbols with language-specific data

Milestone Checkpoints:

After implementing the data model:

1. **Document Operations Test:** Run `npm test src/data-model/document.test.ts` - should show successful document storage, change application, and UTF-16 position calculations
2. **Symbol Resolution Test:** Create a small TypeScript file, build symbol table, verify symbol lookup returns correct declaration locations
3. **Manual Verification:** Use the document store to track a file with incremental changes, confirm version numbers increment correctly and content stays synchronized
4. **Position Arithmetic:** Test position-to-offset conversion with files containing emoji and multi-byte characters - positions should align with LSP client expectations

Common Integration Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Position misalignment with editor	UTF-16 vs UTF-8 confusion	Compare position calculations with client	Use proper UTF-16 code unit arithmetic
Symbol not found errors	Scope chain broken	Trace parent scope relationships	Ensure scopes form proper tree structure
Memory leaks with large files	Documents not released	Monitor Map size growth	Implement LRU eviction and proper cleanup
Stale completion items	Symbol table not updated	Check document version numbers	Rebuild symbols on document changes

Transport Layer Design

Milestone(s): This section corresponds primarily to Milestone 1 (JSON-RPC & Initialization), providing the foundational communication infrastructure that all subsequent milestones depend on for client-server interaction.

The transport layer serves as the foundation of our LSP server, establishing the communication channel between the language client (editor) and our server process. Think of it as the postal service for our language server - it handles the envelope format, message delivery, and routing, ensuring that requests from the editor reach the right handlers and responses make their way back reliably.

Unlike typical web services that communicate over HTTP, LSP servers use a more direct approach: they communicate through standard input and output streams (stdin/stdout) with their parent editor process. This design choice makes LSP servers lightweight and easy to spawn as child processes, but it introduces unique challenges around message framing and parsing that we must address carefully.

The transport layer's primary responsibility is bridging the gap between the raw byte streams from stdin/stdout and the structured JSON-RPC messages that our protocol handler expects. It must detect message boundaries in a continuous stream of data, parse JSON content safely, and route completed messages to the appropriate handlers based on their method names.

JSON-RPC Message Framing

The most critical aspect of LSP communication is understanding how messages are framed over the stdin/stdout streams. Unlike HTTP where connection boundaries naturally separate requests, or WebSockets where frames have built-in length indicators, stdin/stdout provides a continuous stream of bytes with no inherent message boundaries.

LSP solves this problem by borrowing the Content-Length framing approach from HTTP headers. Each JSON-RPC message is preceded by a header section that specifies the exact byte length of the following JSON payload. Think of it like a shipping label that tells you exactly how big the package is before you start unwrapping it.

The message format follows this precise structure:

Component	Format	Description
Content-Length Header	<code>Content-Length: <number>\r\n</code>	Specifies the byte count of the JSON payload
Header Separator	<code>\r\n</code>	Empty line separating headers from payload (HTTP-style)
JSON Payload	Raw JSON bytes	The actual JSON-RPC message content

For example, a complete framed message looks like this in the byte stream:

```
Content-Length: 58\r\n\r\n{"jsonrpc":"2.0","id":1,"method":"initialize","params":{}}
```

The framing parser must handle several challenging scenarios that arise from the streaming nature of the communication. Unlike reading a complete file, stdin data arrives in unpredictable chunks that may split messages, headers, or even individual JSON tokens across multiple read operations.

Partial Message Handling Algorithm:

1. Maintain a persistent buffer that accumulates incoming data chunks across multiple read operations
2. Scan the buffer for the Content-Length header pattern, handling cases where the header spans multiple chunks
3. Parse the length value and validate that it's a positive integer within reasonable bounds (prevent memory exhaustion attacks)
4. Check if enough bytes are available in the buffer to read the complete payload (header + separator + content)
5. If insufficient data is available, wait for the next chunk and append it to the buffer
6. Once a complete message is available, extract the JSON payload and remove the processed bytes from the buffer
7. Parse the JSON content and validate that it conforms to the JSON-RPC 2.0 specification
8. Repeat the process to scan for additional complete messages in the remaining buffer

This algorithm must handle edge cases like headers split across chunk boundaries, malformed Content-Length values, and JSON payloads that contain embedded newlines or special characters.

Message Parsing State Machine:

Current State	Input Data	Next State	Action Taken
WaitingForHeader	Data chunk	ParsingHeader	Append to buffer, scan for Content-Length
ParsingHeader	Complete header	WaitingForPayload	Extract length, validate bounds
ParsingHeader	Incomplete header	ParsingHeader	Continue accumulating header data
WaitingForPayload	Sufficient bytes	MessageComplete	Extract JSON, validate, and dispatch
WaitingForPayload	Insufficient bytes	WaitingForPayload	Wait for more data chunks
MessageComplete	Any data	WaitingForHeader	Reset state, process next message

The parser must be particularly careful about encoding issues. LSP specifies UTF-8 encoding for all message content, but the Content-Length header refers to byte count, not character count. This distinction becomes critical when dealing with non-ASCII characters that occupy multiple bytes in UTF-8 encoding.

Design Insight: The Content-Length framing approach, while borrowed from HTTP, creates unique challenges in a streaming context. Unlike HTTP servers that read a complete request before processing, LSP servers must handle incremental data arrival while maintaining message boundaries. This requires careful buffer management and state tracking.

Content-Length Calculation and Validation:

The transport layer must validate Content-Length values to prevent resource exhaustion attacks and detect protocol violations early. Reasonable bounds checking prevents malicious clients from requesting gigabyte-sized message allocations.

Validation Check	Acceptable Range	Error Handling
Minimum Length	>= 2 bytes	Reject with protocol error
Maximum Length	<= 16MB	Reject with protocol error
Numeric Format	Valid integer	Reject with parse error
Character Encoding	ASCII digits only	Reject with encoding error

Message Dispatching

Once messages are successfully parsed and validated, the transport layer must route them to appropriate handlers based on their JSON-RPC method names and message types. Think of this as a switchboard operator who examines each incoming call and connects it to the right department.

The JSON-RPC specification defines three distinct message types, each requiring different handling approaches:

Message Type	Identification	Response Required	Handler Signature
Request	Has <code>id</code> and <code>method</code> fields	Yes - must send response	<code>(params: any) => Promise<any></code>
Notification	Has <code>method</code> , no <code>id</code> field	No - fire-and-forget	<code>(params: any) => void</code>
Response	Has <code>id</code> , <code>result</code> or <code>error</code>	No - completes pending request	Internal promise resolution

The message dispatcher maintains a registry of handler functions mapped to method names. During server initialization, each component registers its handlers for specific LSP methods. For example, the document manager registers handlers for `textDocument/didOpen`, `textDocument/didChange`, and `textDocument/didClose`, while feature providers register handlers for `textDocument/completion`, `textDocument/hover`, and similar methods.

Handler Registration Interface:

Method Name	Parameters	Returns	Description
<code>onRequest</code>	method: string, handler: Function	void	Registers a handler for JSON-RPC requests that require responses
<code>onNotification</code>	method: string, handler: Function	void	Registers a handler for JSON-RPC notifications (no response)
<code>sendRequest</code>	method: string, params: any	Promise	Sends a request to the client and returns a promise for the response
<code>sendNotification</code>	method: string, params: any	void	Sends a notification to the client (fire-and-forget)

The dispatcher must handle several complex scenarios that arise during message routing. Method names that don't match any registered handler should generate appropriate JSON-RPC error responses rather than silently failing. Handlers that throw exceptions must be caught and converted to proper error responses to prevent the entire server from crashing.

Message Processing Algorithm:

1. Validate that the parsed JSON conforms to the JSON-RPC 2.0 specification (required `jsonrpc` field with value `"2.0"`)
2. Determine the message type by examining the presence of `id`, `method`, `result`, and `error` fields
3. For requests: look up the handler in the request registry using the method name
4. For notifications: look up the handler in the notification registry using the method name
5. For responses: match the `id` field against pending outgoing requests and resolve the corresponding promise
6. Execute the handler function with the message parameters, catching any exceptions
7. For requests, serialize the handler's return value or exception as a JSON-RPC response message
8. Send the response through the output stream using proper Content-Length framing

The dispatcher must maintain a map of pending outgoing requests to handle responses from the client. When our server sends a request to the client (such as `window/showMessage` or `client/registerCapability`), it stores a promise resolver in the pending requests map keyed by the request ID. When the client's response arrives, the dispatcher locates the corresponding promise and resolves or rejects it appropriately.

Critical Insight: JSON-RPC is bidirectional - both client and server can send requests to each other. The transport layer must handle incoming responses to our outgoing requests, not just route incoming requests from the client. This bidirectional nature is essential for LSP features like client capability registration and user interface interactions.

Error Response Generation:

When handlers encounter errors or when invalid messages are received, the transport layer must generate appropriate JSON-RPC error responses following the specification's error code conventions.

Error Scenario	Error Code	Error Message	When to Use
Method Not Found	-32601	Method not found	Handler not registered for method
Invalid Params	-32602	Invalid params	Handler rejects parameter format
Internal Error	-32603	Internal error	Handler throws unexpected exception
Parse Error	-32700	Parse error	JSON syntax is invalid
Invalid Request	-32600	Invalid Request	Missing required JSON-RPC fields

Architecture Decisions

The transport layer design involves several critical architecture decisions that impact performance, reliability, and maintainability. Each decision represents a trade-off between different quality attributes that must be carefully evaluated.

Decision: Streaming vs Buffered Message Processing

- **Context:** LSP servers receive data through stdin in unpredictable chunks that may contain partial messages, complete messages, or multiple messages. We must decide whether to process messages as soon as they arrive or buffer them for batch processing.
- **Options Considered:**
 1. Immediate streaming - process each chunk as it arrives
 2. Message buffering - accumulate complete messages before processing
 3. Hybrid approach - stream parsing with message-level dispatch
- **Decision:** Hybrid approach with streaming parse and message-level dispatch
- **Rationale:** Streaming parsing provides low latency for interactive features like completion, while message-level dispatch ensures atomic handling of complete requests. This prevents race conditions where a handler sees partial request data.
- **Consequences:** Requires more complex buffer management but provides optimal responsiveness while maintaining consistency guarantees.

Approach	Latency	Memory Usage	Complexity	Chosen?
Immediate Streaming	Lowest	Lowest	High (partial message handling)	No
Message Buffering	Higher	Higher	Low	No
Hybrid	Low	Moderate	Moderate	✓

Decision: Synchronous vs Asynchronous Handler Execution

- **Context:** LSP handlers may perform expensive operations like parsing large files or analyzing complex codebases. We must decide whether to execute handlers synchronously or asynchronously to prevent blocking other requests.
- **Options Considered:**
 1. Synchronous execution - simple but blocks other requests
 2. Full async with worker threads - complex but highly concurrent
 3. Async with cooperative scheduling - balanced approach
- **Decision:** Async with cooperative scheduling using promises
- **Rationale:** LSP clients expect quick responses for interactive features. Async execution with promise-based handlers allows long-running operations to yield control while maintaining simple programming model. TypeScript's native Promise support makes this natural.
- **Consequences:** Handlers must be written as async functions, but the server remains responsive during expensive operations like workspace analysis.

Approach	Responsiveness	Simplicity	Resource Usage	Chosen?
Synchronous	Poor (blocking)	High	Low	No
Full Async/Threading	Excellent	Low	High	No
Promise-based Async	Good	Moderate	Moderate	✓

Decision: Error Isolation and Recovery Strategy

- **Context:** Handler functions may throw exceptions due to malformed input, file system errors, or bugs. We must decide how to isolate these failures and whether to attempt recovery or crash the entire server.
- **Options Considered:**
 1. Global error handling - catch all exceptions at transport level
 2. Per-handler isolation - isolate each handler's errors
 3. Graceful degradation - continue serving other features after errors
- **Decision:** Per-handler isolation with graceful degradation
- **Rationale:** LSP servers should remain available even when individual features fail. A crash in the hover provider shouldn't break completion or diagnostics. Isolating errors per handler and returning proper JSON-RPC error responses keeps the client informed while maintaining service availability.
- **Consequences:** Requires careful error boundary design and comprehensive logging, but provides superior reliability and user experience.

Approach	Reliability	User Experience	Debugging	Chosen?
Global Error Handling	Poor (crash)	Poor (total failure)	Simple	No
Per-handler Isolation	Good	Good (partial failure)	Moderate	✓
No Error Handling	Very Poor	Very Poor	Hard	No

Decision: Message Ordering and Concurrency Control

- **Context:** Multiple LSP requests may arrive simultaneously, and some operations like document synchronization must be processed in order while others like completion can be processed concurrently.
- **Options Considered:**
 1. Strict sequential processing - simple but slow
 2. Full concurrency - fast but risks race conditions
 3. Per-document ordering with cross-document concurrency
- **Decision:** Per-document ordering with cross-document concurrency
- **Rationale:** Document synchronization events (`didOpen`, `didChange`, `didClose`) must be processed in order per document to maintain consistent state. However, requests for different documents can be processed concurrently for better performance. Read-only operations like completion can run concurrently even on the same document.
- **Consequences:** Requires document-level locking and request classification, but provides optimal performance while preventing race conditions.

Common Pitfalls in Transport Layer Implementation:

⚠ **Pitfall: Incorrect Content-Length Calculation** Many implementations fail to correctly calculate byte lengths when dealing with Unicode characters. The Content-Length header specifies byte count, not character count, which differs for non-ASCII characters in UTF-8 encoding. For example, the string `"café"` contains 4 characters but 5 bytes (the `é` requires 2 bytes). Always use byte-based length calculation:

```
Buffer.byteLength(jsonString, 'utf8')
```

⚠ **Pitfall: Incomplete Buffer Management** Failing to properly handle partial messages across read operations leads to corrupted or lost messages. The parser must maintain a persistent buffer and handle cases where Content-Length headers, JSON payloads, or even individual tokens span multiple chunks. Never assume that each read operation delivers complete messages.

⚠ **Pitfall: Missing Error Response Generation** When handlers throw exceptions or unknown methods are called, failing to send proper JSON-RPC error responses leaves the client waiting indefinitely for a response. Always wrap handler execution in try-catch blocks and generate appropriate error responses with standard error codes.

⚠ **Pitfall: Blocking Handler Execution** Executing long-running operations synchronously in request handlers blocks the entire message processing pipeline, making the server unresponsive. Always use async handlers for

operations that might take significant time, such as file I/O, parsing, or analysis.

⚠ Pitfall: Race Conditions in Document State Processing document synchronization events (`didOpen`, `didChange`, `didClose`) concurrently can lead to inconsistent document state where changes are applied out of order. Implement per-document ordering to ensure these critical events are processed sequentially.



Implementation Guidance

The transport layer requires careful implementation to handle the streaming nature of stdin/stdout communication while maintaining robust error handling and message routing capabilities.

A. Technology Recommendations:

Component	Simple Option	Advanced Option
Stream Processing	Node.js readline + manual parsing	Custom streaming parser with state machine
JSON Processing	JSON.parse() with try-catch	JSON streaming parser for large messages
Message Routing	Simple object map of handlers	Event emitter pattern with middleware
Error Handling	Basic try-catch with logging	Structured error types with recovery strategies
Buffer Management	String concatenation	Buffer pooling with memory limits

B. Recommended File Structure:

```

src/
  transport/
    jsonrpc.ts           ← JSON-RPC message types and validation
    message-framer.ts   ← Content-Length parsing and message extraction
    message-dispatcher.ts ← Handler registration and message routing
    stream-transport.ts  ← Main transport class coordinating all components
    transport-types.ts   ← Common interfaces and types
    server.ts            ← Main LSP server that uses transport layer
  
```

C. Infrastructure Starter Code:

Here's a complete message framer implementation that handles the complex Content-Length parsing:

```
export class MessageFramer {

    private buffer: string = '';

    private readonly CONTENT_LENGTH_PATTERN = /Content-Length: (\d+)\r?\n/i;

    private readonly HEADER_SEPARATOR = '\r\n\r\n';

    handleData(chunk: string): JsonRpcMessage[] {

        this.buffer += chunk;

        const messages: JsonRpcMessage[] = [];

        while (true) {

            const message = this.tryExtractMessage();

            if (!message) break;

            messages.push(message);

        }

        return messages;
    }

    private tryExtractMessage(): JsonRpcMessage | null {

        // Find Content-Length header

        const headerMatch = this.buffer.match(this.CONTENT_LENGTH_PATTERN);

        if (!headerMatch) return null;

        const contentLength = parseInt(headerMatch[1], 10);

        if (contentLength <= 0 || contentLength > 16 * 1024 * 1024) {

            throw new Error(`Invalid Content-Length: ${contentLength}`);
        }
    }
}
```

```
}

// Find header separator

const separatorIndex = this.buffer.indexOf(this.HEADER_SEPARATOR);

if (separatorIndex === -1) return null;

const headerEndIndex = separatorIndex + this.HEADER_SEPARATOR.length;

const totalMessageLength = headerEndIndex + contentLength;

// Check if we have the complete message

if (this.buffer.length < totalMessageLength) return null;

// Extract and parse JSON payload

const jsonPayload = this.buffer.substring(headerEndIndex, totalMessageLength);

this.buffer = this.buffer.substring(totalMessageLength);

try {

    return JSON.parse(jsonPayload) as JsonRpcMessage;

} catch (error) {

    throw new Error(`Invalid JSON in message payload: ${error.message}`);

}

frameMessage(message: JsonRpcMessage): string {

    const jsonContent = JSON.stringify(message);

    const contentLength = Buffer.byteLength(jsonContent, 'utf8');

    return `Content-Length: ${contentLength}\r\n\r\n${jsonContent}`;
}
```

```
}
```

```
}
```

Here's a complete JSON-RPC type system:

```
export interface JsonRpcMessage {  
    jsonrpc: '2.0';  
  
    id?: string | number;  
  
    method?: string;  
  
    params?: any;  
  
    result?: any;  
  
    error?: {  
  
        code: number;  
  
        message: string;  
  
        data?: any;  
  
    };  
}  
  
export interface JsonRpcRequest extends JsonRpcMessage {  
  
    method: string;  
  
    params?: any;  
}  
  
export interface JsonRpcNotification extends JsonRpcMessage {  
  
    method: string;  
  
    params?: any;  
}  
  
export interface JsonRpcResponse extends JsonRpcMessage {  
  
    id: string | number;  
  
    result?: any;  
  
    error?: {  
  
        code: number;  
    };  
}
```

```

    message: string;

    data?: any;

};

}

export const JSONRPC_VERSION = '2.0';

export const METHOD_NOT_FOUND = -32601;

export const INVALID_PARAMS = -32602;

export const INTERNAL_ERROR = -32603;

export const PARSE_ERROR = -32700;

export const INVALID_REQUEST = -32600;

export function isRequest(message: JsonRpcMessage): message is JsonRpcRequest {

    return message.method !== undefined && message.id !== undefined;
}

export function isNotification(message: JsonRpcMessage): message is JsonRpcNotification {

    return message.method !== undefined && message.id === undefined;
}

export function isResponse(message: JsonRpcMessage): message is JsonRpcResponse {

    return message.id !== undefined && message.method === undefined;
}

```

D. Core Logic Skeleton Code:

```
// message-dispatcher.ts
```

TYPESCRIPT

```
export class MessageDispatcher {

    private requestHandlers = new Map<string, (params: any) => Promise<any>>();

    private notificationHandlers = new Map<string, (params: any) => void>();

    private pendingRequests = new Map<string | number, {
        resolve: (result: any) => void;
        reject: (error: any) => void;
    }>();

    private nextRequestId = 1;

    onRequest(method: string, handler: (params: any) => Promise<any>): void {
        // TODO 1: Validate that method is a non-empty string
        // TODO 2: Store handler in requestHandlers map
        // TODO 3: Log handler registration for debugging
    }

    onNotification(method: string, handler: (params: any) => void): void {
        // TODO 1: Validate that method is a non-empty string
        // TODO 2: Store handler in notificationHandlers map
        // TODO 3: Log handler registration for debugging
    }

    async processMessage(message: JsonRpcMessage): Promise<JsonRpcMessage | null> {
        // TODO 1: Validate message has required jsonrpc field with value "2.0"
        // TODO 2: Use isRequest/isNotification/isResponse to determine message type
        // TODO 3: For requests: look up handler, execute with error handling, return response
        // TODO 4: For notifications: look up handler, execute with error handling, return
        null
        // TODO 5: For responses: match against pendingRequests and resolve/reject promise
    }
}
```

```
// TODO 6: Generate appropriate error responses for unhandled methods or exceptions

// Hint: Wrap handler execution in try-catch and convert exceptions to JSON-RPC errors

}

sendRequest(method: string, params: any): Promise<any> {

    // TODO 1: Generate unique request ID using nextRequestId

    // TODO 2: Create JsonRpcRequest message with method, params, and ID

    // TODO 3: Create Promise and store resolve/reject in pendingRequests map

    // TODO 4: Send framed message to stdout

    // TODO 5: Return the promise (will be resolved when response arrives)

}

sendNotification(method: string, params: any): void {

    // TODO 1: Create JsonRpcNotification message with method and params

    // TODO 2: Send framed message to stdout

    // TODO 3: No response expected, so return immediately

}

private createErrorResponse(id: string | number | undefined, code: number, message: string): JsonRpcResponse {

    // TODO 1: Handle case where id is undefined (use null for parse errors)

    // TODO 2: Create JsonRpcResponse with error field containing code and message

    // TODO 3: Ensure jsonrpc field is set to "2.0"

}

}
```

```
// stream-transport.ts
```

TYPESCRIPT

```
export class StreamTransport {

    private framer = new MessageFramer();

    private dispatcher = new MessageDispatcher();

    start(): void {
        // TODO 1: Set up stdin to receive data in utf8 encoding
        // TODO 2: Register data event handler that calls handleData
        // TODO 3: Register error event handlers for stdin/stdout
        // TODO 4: Set up proper shutdown handling for SIGINT/SIGTERM
    }

    private async handleData(chunk: string): Promise<void> {
        // TODO 1: Pass chunk to framer.handleData() to extract complete messages
        // TODO 2: For each extracted message, call dispatcher.processMessage()
        // TODO 3: If processMessage returns a response, send it via sendMessage()
        // TODO 4: Catch and log any errors without crashing the server
        // Hint: Use Promise.all() if processing multiple messages concurrently
    }

    setupHandlers(): void {
        // TODO 1: Register initialize request handler
        // TODO 2: Register shutdown request handler
        // TODO 3: Register exit notification handler
        // TODO 4: Register document synchronization notification handlers
        // TODO 5: Register language feature request handlers
        // Hint: This will be called by the main LSP server during startup
    }
}
```

```

private sendMessage(message: JsonRpcMessage): void {

    // TODO 1: Use framer.frameMessage() to add Content-Length header

    // TODO 2: Write framed message to stdout

    // TODO 3: Handle write errors gracefully

}

}

```

E. Language-Specific Hints:

- Use `process.stdin` and `process.stdout` for stdio communication in Node.js
- Set `process.stdin.setEncoding('utf8')` to handle Unicode correctly
- Use `Buffer.byteLength(string, 'utf8')` for accurate Content-Length calculation
- Handle `process.stdin.on('data', handler)` for streaming input
- Use `async/await` consistently for all handler functions to prevent blocking
- Consider using `process.stdin.setRawMode(false)` to ensure line buffering works correctly

F. Milestone Checkpoint:

After implementing the transport layer, verify correct operation:

Command to run:

```

npm test -- --grep "transport"                                     BASH

node dist/server.js

```

Expected behavior:

1. Server starts and waits for input on stdin
2. Send a test initialize request:

```

Content-Length: 88

{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"capabilities":[]}}

```

3. Server should respond with an initialize response containing `ServerCapabilities`
4. Check that Content-Length header in response matches actual payload byte count
5. Verify that malformed messages generate appropriate error responses

Signs of problems:

- Server hangs without responding → Check message framing and handler registration
- Responses have incorrect Content-Length → Check byte length calculation for UTF-8

- Server crashes on invalid input → Add error handling around JSON parsing and handler execution
- Messages are split or corrupted → Verify buffer management in message framer

G. Debugging Tips:

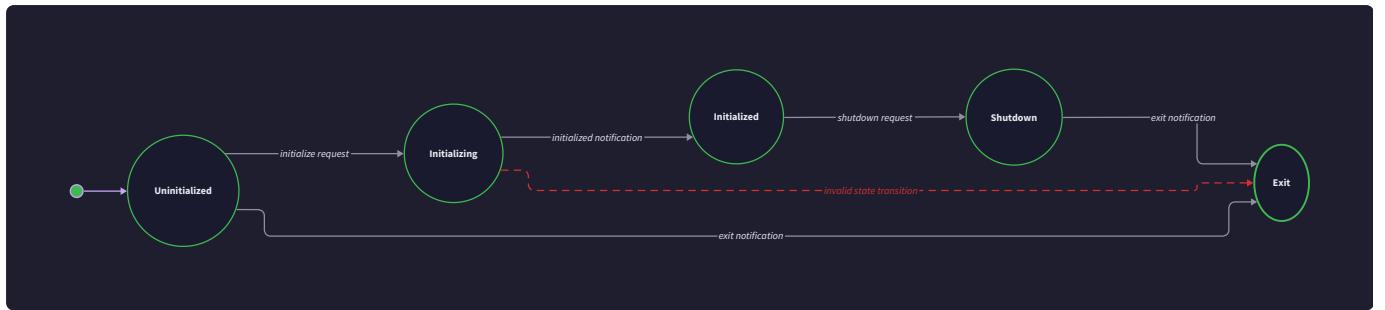
Symptom	Likely Cause	How to Diagnose	Fix
Server hangs on startup	stdin not configured properly	Check if <code>stdin.setEncoding()</code> is called	Set encoding and register data handlers
Messages are truncated	Incorrect Content-Length calculation	Log actual vs calculated byte lengths	Use <code>Buffer.byteLength()</code> for UTF-8 strings
JSON parse errors	Buffer boundary issues	Log raw message content before parsing	Fix message boundary detection in framer
Handler exceptions crash server	Missing error handling	Add logging in message dispatcher	Wrap handler calls in try-catch blocks
Responses never reach client	stdout not flushed	Check if <code>stdout.write()</code> completes	Ensure proper message framing and output

Protocol Handler Design

Milestone(s): This section corresponds primarily to Milestone 1 (JSON-RPC & Initialization), establishing the core protocol management layer that coordinates initialization handshakes, capability negotiation, and server lifecycle management throughout all subsequent milestones.

Think of the **Protocol Handler** as the diplomatic ambassador between your language server and the outside world. Just as an ambassador manages formal communications, establishes diplomatic relations, and ensures both parties understand what services they can provide to each other, the Protocol Handler manages the LSP protocol layer, orchestrates the initialization handshake, and coordinates the ongoing relationship between client and server. While the Transport Layer handles the mechanics of message delivery (like the postal service), the Protocol Handler understands the meaning and timing of those messages (like the embassy that interprets and responds to diplomatic communications).

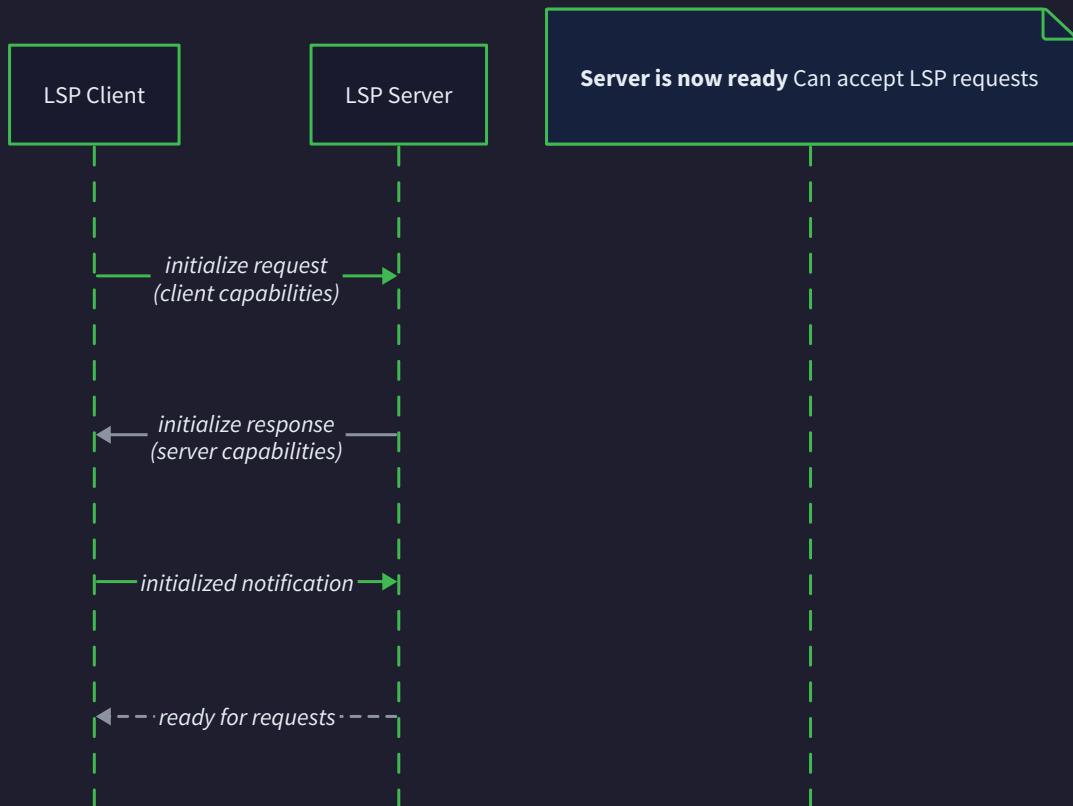
The Protocol Handler serves as the **control plane** for the entire LSP server. It doesn't parse your source code or generate completions—that's the job of specialized components downstream. Instead, it ensures the server follows LSP protocol rules, transitions through proper states, advertises accurate capabilities, and coordinates the overall request-response lifecycle. This separation of concerns allows the language-specific logic to focus purely on analysis while the protocol logic handles the standardized communication patterns that work across all editors and languages.



Initialization Handshake

The **initialization handshake** is the formal diplomatic ceremony that establishes the working relationship between client and server. Think of it like the opening negotiations of a business partnership—both parties need to identify themselves, state what they're capable of doing, agree on working terms, and formally seal the deal before any real work can begin.

The LSP initialization follows a strict two-phase protocol designed to prevent race conditions and ensure both parties are fully synchronized before language requests begin flowing. Unlike a simple "hello" message, this handshake involves **capability negotiation** where each party declares their abilities and the intersection determines what features will be available during the session.



The initialization sequence consists of four distinct phases, each with specific responsibilities and failure modes:

Phase	Message	Direction	Purpose	Required Response	Error Handling
1. Initiation	<code>initialize</code> request	Client → Server	Client declares capabilities and requests server info	<code>InitializeResult</code> with server capabilities	Return error response, do not transition state
2. Confirmation	<code>InitializeResult</code> response	Server → Client	Server declares capabilities and confirms readiness	Client processes capabilities	Client may disconnect if capabilities insufficient
3. Acknowledgment	<code>initialized</code> notification	Client → Server	Client confirms initialization complete	No response required	Log warning, continue operation
4. Ready State	Normal operations	Bidirectional	All LSP features now available	Per-method responses	Standard error handling applies

The **initialize request** carries comprehensive client information that determines how the server should behave throughout the session. The client provides its process ID (for lifecycle management), root workspace URI (for multi-file analysis), and detailed capability declarations that specify which LSP features it supports.

Client Capability	Type	Purpose	Server Decision Impact
<code>textDocument.completion.completionItem.snippetSupport</code>	boolean	Can client render snippet completions	Determines whether to return snippet vs plain text
<code>textDocument.hover.contentFormat</code>	string[]	Supported markup formats (markdown, plaintext)	Chooses response format for hover content
<code>workspace.workspaceFolders</code>	boolean	Can client provide multiple workspace roots	Enables workspace-wide symbol resolution
<code>window.showMessage</code>	object	Can client display server messages	Determines error reporting strategy

The server's **InitializeResult** response contains the `ServerCapabilities` object that formally declares which LSP features this server implementation supports. This isn't just a boolean checklist—many capabilities include configuration objects that specify exactly how features behave:

Server Capability	Configuration Object	Behavior Specification
<code>completionProvider</code>	<pre>{ triggerCharacters: ['. ', ':'], resolveProvider: true }</pre>	Which characters auto-trigger completion, whether completion items can be resolved for additional details
<code>textDocumentSync</code>	<pre>{ openClose: true, change: 2 }</pre>	Whether server wants open/close notifications, sync mode (1=full, 2=incremental)
<code>codeActionProvider</code>	<pre>{ codeActionKinds: ['quickfix', 'refactor'] }</pre>	Which types of code actions this server provides
<code>definitionProvider</code>	<code>true</code>	Simple boolean—server supports go-to-definition

Design Insight: The capability objects use **progressive disclosure**—simple features are just booleans (`definitionProvider: true`) while complex features include rich configuration objects. This allows the protocol to evolve without breaking existing implementations.

Consider a concrete initialization scenario: VS Code opens a TypeScript project and launches our LSP server. VS Code sends an initialize request declaring it supports snippet completions, markdown hover content, and workspace folders. Our server examines these capabilities and responds with its own capabilities: completion provider with trigger characters `['.', ':']`, hover provider supporting markdown, definition provider, but no code action provider. VS Code receives this response, notes that code actions won't be available, then sends the `initialized` notification. At this point, the server transitions to the ready state and VS Code begins sending `textDocument/didOpen` notifications for each open file.

Decision: Two-Phase Initialization Protocol

- **Context:** LSP needs to prevent race conditions where language requests arrive before the server is fully configured
- **Options Considered:**
 1. Single initialize request with immediate ready state
 2. Two-phase initialize + initialized handshake
 3. Three-phase handshake with separate capability negotiation
- **Decision:** Two-phase initialize + initialized handshake
- **Rationale:** Single-phase allows race conditions where client sends requests before server is ready. Three-phase adds unnecessary complexity. Two-phase provides clean synchronization point while remaining simple.
- **Consequences:** Servers must handle the uninitialized state correctly and reject requests until `initialized` notification arrives. Clients must wait for `InitializeResult` before sending the `initialized` notification.

The `handleInitialize` method serves as the server's opportunity to examine client capabilities and make configuration decisions that will affect the entire session. The implementation must validate the client's request, configure internal components based on declared capabilities, and return accurate capability declarations.

Initialize Handler Responsibility	Implementation Detail	Error Conditions
Validate client process ID	Store PID for lifecycle management	Invalid or missing PID → continue with warning
Process workspace configuration	Configure document search paths and URI resolution	Invalid workspace URI → use null workspace
Examine client capabilities	Configure feature providers based on client support	Missing capability declarations → assume minimal support
Initialize internal components	Create DocumentManager, LanguageEngine instances	Component initialization failure → return error response
Return server capabilities	Build ServerCapabilities object with accurate feature list	Never return capabilities the server cannot provide

Capability Negotiation

Capability negotiation is the technical contract establishment phase where client and server agree on exactly which LSP features will be active during the session and how they will behave. Think of it as two software systems comparing their feature matrices and finding the intersection—only the features both parties support can be used, and the specific configuration details determine the exact behavior.

This negotiation is **asymmetric** by design. The client declares what it can handle (can it render snippet completions? does it support markdown in hover responses?), while the server declares what it can provide (does it support go-to-definition? can it provide code actions?). The intersection of these capabilities determines the effective feature set, but each side has different responsibilities in making this work.

The capability negotiation solves a critical **versioning problem** in IDE integrations. Different editors support different subsets of LSP features, and those features evolve over time. Without explicit negotiation, a server might return snippet completions to an editor that can only render plain text, or provide markdown hover content to a client that expects plain text, resulting in broken user experiences.

Negotiation Pattern	Client Responsibility	Server Responsibility	Failure Mode
Feature Availability	Declare supported features in client capabilities	Only advertise features in server capabilities if implemented	Server advertises unsupported feature → client requests fail
Content Format	Declare supported formats (markdown, plaintext)	Choose appropriate format based on client support	Server uses unsupported format → client displays garbage
Behavioral Configuration	Declare behavioral preferences (snippet support, resolve provider)	Configure feature behavior to match client expectations	Mismatch → features work but UX is degraded
Extension Capabilities	Declare vendor-specific extensions	Recognize and conditionally enable extensions	Unknown extensions → ignored safely

Document synchronization mode illustrates the complexity of capability negotiation. The server can support different synchronization strategies, each with different performance characteristics and implementation complexity:

Sync Mode	Value	Client Requirement	Server Implementation	Performance Trade-off
None	0	Client never sends document content	Server must read files from disk	Low network usage, high disk I/O, stale content risk
Full	1	Client sends entire document on every change	Server replaces entire document content	High network usage, simple implementation
Incremental	2	Client sends only changed ranges	Server applies edits to existing content	Low network usage, complex edit application

Decision: Server-Driven Capability Negotiation

- **Context:** The server needs to know which features to enable and how to format responses, while the client needs to know which features are available
- **Options Considered:**
 1. Client-driven negotiation where client requests specific features
 2. Server-driven negotiation where server declares available features
 3. Bidirectional negotiation with multiple rounds
- **Decision:** Server-driven negotiation with client capabilities as input
- **Rationale:** Server knows its own implementation capabilities better than client can guess. Client capabilities serve as constraints on server behavior rather than requests for features. Single-round negotiation keeps protocol simple.
- **Consequences:** Servers must accurately advertise only implemented features. Clients must handle cases where desired features are unavailable.

The **completion provider capability** demonstrates how complex capabilities work in practice. Rather than a simple boolean, completion providers include rich configuration that affects both client and server behavior:

Completion Configuration	Type	Purpose	Client Behavior	Server Behavior
<code>triggerCharacters</code>	<code>string[]</code>	Characters that auto-trigger completion	Show completion popup immediately when user types these	Send completion request automatically, expect high volume
<code>allCommitCharacters</code>	<code>string[]</code>	Characters that accept completion and continue typing	Apply selected completion then insert the typed character	Include commit characters in CompletionItem responses
<code>resolveProvider</code>	<code>boolean</code>	Can server provide additional details for completion items	Enable lazy loading of documentation and imports	Implement textDocument/completionItem/resolve handler
<code>workDoneProgress</code>	<code>boolean</code>	Can server report progress for long-running completion	Show progress indicator for slow completions	Send progress notifications during analysis

Consider a hover capability negotiation scenario: The client declares it supports both `markdown` and `plaintext` content formats in its hover capability. Our server is capable of generating rich markdown documentation with code examples and links, but also has a fallback plain text mode. During initialization, the server examines the client's `contentFormat` array, sees that markdown is supported, and configures its hover provider to generate markdown responses. If the client had only declared plaintext support, the server would configure its hover provider to strip formatting and return plain text.

The capability negotiation also handles **progressive enhancement** scenarios where newer features are added to LSP over time. Clients and servers can safely ignore capabilities they don't understand, allowing newer servers to work with older editors and vice versa.

Progressive Enhancement Pattern	Example	Backward Compatibility Strategy
New capability with fallback	<code>codeAction.dataSupport</code> for lazy resolution	Server checks if supported; if not, includes all data in initial response
New content type	<code>CompletionItem.labelDetails</code> for rich completion display	Server provides if client supports, falls back to simple label
New request method	<code>textDocument/prepareRename</code> for rename validation	Server advertises only if client can handle, uses immediate rename otherwise
Behavioral flag	<code>workspace.workspaceEdit.resourceOperations</code> for file operations	Server limits edit responses to operations client can perform

Server Lifecycle Management

Server lifecycle management orchestrates the birth, life, and death of the LSP server process, ensuring clean transitions between states and proper resource cleanup. Think of it as the operating system for your language server—it manages the fundamental state transitions that allow all other components to operate safely, coordinates startup and shutdown sequences, and ensures the server never gets stuck in inconsistent states.

The LSP server lifecycle is a **finite state machine** with five distinct states, each with specific allowed operations and transition rules. Unlike a simple on/off switch, this state machine enforces the protocol invariants that keep client and server synchronized throughout their relationship.

Server State	Allowed Operations	Incoming Requests	Resource Status	Exit Conditions
Uninitialized	Accept initialize request only	initialize → allowed, all others → error	Minimal resources allocated	Receive initialize request
Initializing	Process initialize request	initialize → error, all others → error	Components initializing	Send InitializeResult response
Initialized	All LSP operations	All <code>textDocument/</code> , <code>workspace/</code> requests	All components active	Receive shutdown request
ShuttingDown	Cleanup only	shutdown → error, all others → error	Components cleaning up	Send shutdown response
Exited	None	exit → terminate process	All resources released	Process termination

The **uninitialized state** is the server's initial condition when the process starts but before any LSP communication has occurred. In this state, the server has allocated minimal resources—perhaps just the transport layer and message dispatcher—but hasn't created language-specific components like the document manager or AST parser. This lazy initialization prevents resource waste and allows the server to configure itself based on the client's declared capabilities.

Design Insight: The uninitialized state serves as a **configuration barrier**. All language-specific initialization must wait until the server knows which features the client supports, preventing the server from wasting resources on unused capabilities.

The **initialization transition** occurs when the server receives and successfully processes the initialize request. During this transition, the server examines client capabilities, creates and configures all internal components (DocumentManager, LanguageEngine, feature providers), establishes any external connections (file watchers, database connections), and prepares to handle language requests. This is the most complex state transition because it involves **dependency injection** throughout the component hierarchy.

Initialization Phase	Component Creation	Configuration Source	Failure Recovery
Transport Setup	Already complete	Command line args	Process exit if failed
Capability Analysis	Capability analyzer	InitializeParams.capabilities	Use conservative defaults
Core Component Creation	DocumentManager, LanguageEngine	Server configuration + client capabilities	Return error response
Feature Provider Setup	Completion, Hover, Definition providers	ServerCapabilities decisions	Disable failed features
Resource Allocation	File watchers, caches	Workspace configuration	Continue with reduced functionality

Consider a complex initialization scenario: An LSP server starts up and receives an initialize request from VS Code with a large workspace containing 10,000 source files. During initialization, the server must: (1) create a DocumentManager configured for incremental sync, (2) initialize a LanguageEngine with the appropriate parser for the project's language version, (3) set up file watchers for the workspace directories, (4) create feature providers configured for VS Code's capabilities, and (5) optionally start background indexing of the workspace. If the file watcher setup fails due to system limits, the server should continue initialization but log a warning that workspace-wide features may be limited.

The **shutdown sequence** is designed to ensure **graceful degradation** when the client wants to terminate the session. Unlike a sudden process kill, the LSP shutdown protocol gives the server time to complete in-progress operations, save any persistent state, and release resources cleanly.

Decision: Two-Phase Shutdown Protocol

- **Context:** LSP servers may have long-running operations, persistent caches, or external connections that need clean cleanup
- **Options Considered:**
 1. Immediate exit notification with process termination
 2. Two-phase shutdown request + exit notification
 3. Three-phase shutdown with save confirmation
- **Decision:** Two-phase shutdown request followed by exit notification
- **Rationale:** Immediate exit doesn't allow cleanup of persistent state or completion of in-flight operations. Three-phase adds complexity without significant benefit. Two-phase provides cleanup window while keeping protocol simple.
- **Consequences:** Servers must implement cleanup logic in shutdown handler. Clients must wait for shutdown response before sending exit notification.

The shutdown transition involves **ordered component teardown** to prevent resource leaks and data corruption. The server should shut down components in reverse dependency order—feature providers first, then language engine, then document manager—to ensure no component tries to use a resource that has already been cleaned up.

Shutdown Phase	Components Affected	Cleanup Actions	Timeout Handling
Request Rejection	All feature providers	Stop accepting new requests, finish in-progress	Cancel after configured timeout
State Persistence	DocumentManager, LanguageEngine	Save any cacheable analysis results	Skip saves that would block
Resource Cleanup	File watchers, network connections	Close handles, cancel background tasks	Force-close after timeout
Memory Cleanup	All components	Clear large data structures	Let process exit handle cleanup if slow

Error handling during lifecycle transitions requires special consideration because normal error recovery mechanisms may not be available. During initialization, if a critical component fails to start, the server should return an error response to the initialize request and remain in the uninitialized state rather than transitioning to a partially-working initialized state.

Lifecycle Error	Detection	Recovery Strategy	Client Notification
Initialization component failure	Exception during component creation	Return error in InitializeResult	Client receives error response, can retry or exit
Workspace access denied	File system errors during setup	Continue with limited functionality	Return success but log warnings
Shutdown timeout	Component cleanup takes too long	Force termination after timeout	Client may see connection drop instead of clean exit
Crash during operation	Process termination or uncaught exception	No recovery possible	Client detects connection loss, may restart server

The **exit notification** is the final message in the server's lifecycle. Unlike other LSP messages, exit is a **fire-and-forget notification** that should cause immediate process termination with exit code 0. The server should not send a response to exit—the client detects successful termination by observing that the server process has ended cleanly.

Common Pitfalls

⚠ Pitfall: Accepting Requests Before Initialization Many LSP server implementations mistakenly accept `textDocument/completion` or other language requests before receiving the `initialized` notification. This

violates the LSP protocol and can cause race conditions where requests arrive before the server has configured its components based on client capabilities. Always check server state in your message dispatcher and return `serverNotInitialized` errors for requests that arrive too early.

⚠ Pitfall: Inaccurate Capability Advertising Servers sometimes advertise capabilities in their `InitializeResult` that they don't actually implement, causing client requests to fail mysteriously. For example, advertising `definitionProvider: true` but not implementing a `textDocument/definition` handler. The client will send definition requests and receive "method not found" errors, breaking the user experience. Only advertise capabilities you've actually implemented and tested.

⚠ Pitfall: Ignoring Client Capabilities During Feature Implementation A common mistake is implementing features without checking what the client supports. For example, always returning markdown in hover responses without checking if the client declared markdown support in its capabilities. This causes garbage text to appear in editors that only support plain text. Always check client capabilities before choosing response formats.

⚠ Pitfall: Blocking Initialization on Expensive Operations Some servers attempt to index entire large codebases during the initialization request, causing timeouts. The initialize request should complete quickly (under a few seconds) by setting up components and deferring expensive work like workspace indexing to background tasks after the `initialized` notification.

⚠ Pitfall: Improper Shutdown Cleanup Failing to implement proper cleanup in the shutdown handler can leave zombie processes, unclosed file handles, or corrupted cache files. Always implement orderly component teardown and test shutdown scenarios, including abnormal termination cases where the client disconnects without sending shutdown.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
State Machine	Simple enum with switch statements	State pattern with explicit transition validation
Message Routing	Map of method names to handler functions	Decorator pattern with middleware support
Capability Storage	Plain objects with optional properties	Type-safe capability classes with validation
Lifecycle Events	Direct method calls	Event emitter pattern for component coordination
Error Handling	Try-catch with JSON-RPC error responses	Result type pattern with explicit error propagation

Recommended File Structure

```
project-root/
├── src/
|   ├── protocol/
|   |   ├── LspServer.ts           ← main protocol handler
|   |   ├── ProtocolState.ts      ← state machine implementation
|   |   ├── CapabilityNegotiator.ts ← capability analysis logic
|   |   └── LifecycleManager.ts    ← startup/shutdown coordination
|   ├── transport/
|   |   ├── StreamTransport.ts     ← from previous section
|   |   └── MessageDispatcher.ts   ← from previous section
|   ├── document/
|   |   └── DocumentManager.ts    ← document synchronization (next section)
|   └── language/
|       └── LanguageEngine.ts      ← analysis engine (later section)
└── test/
    └── protocol/
        ├── initialization.test.ts  ← test handshake scenarios
        └── lifecycle.test.ts       ← test state transitions
```

Infrastructure Starter Code

ProtocolState.ts - Complete state machine implementation:

```
// Protocol state management with validation and transitions
```

TYPESCRIPT

```
export enum ServerState {

    Uninitialized = 'uninitialized',
    Initializing = 'initializing',
    Initialized = 'initialized',
    ShuttingDown = 'shuttingDown',
    Exited = 'exited'

}

export class ProtocolState {

    private currentState: ServerState = ServerState.Uninitialized;
    private readonly transitions: Map<ServerState, Set<ServerState>>;

    constructor() {

        // Define allowed state transitions

        this.transitions = new Map([
            [ServerState.Uninitialized, new Set([ServerState.Initializing])],
            [ServerState.Initializing, new Set([ServerState.Initialized])],
            [ServerState.Initialized, new Set([ServerState.ShuttingDown])],
            [ServerState.ShuttingDown, new Set([ServerState.Exited])],
            [ServerState.Exited, new Set()] // Terminal state
        ]);
    }

    canTransition(to: ServerState): boolean {

        const allowedTransitions = this.transitions.get(this.currentState);

        return allowedTransitions ? allowedTransitions.has(to) : false;
    }
}
```

```
transition(to: ServerState): void {
    if (!this.canTransition(to)) {
        throw new Error(`Invalid transition from ${this.currentState} to ${to}`);
    }
    this.currentState = to;
}

get state(): ServerState {
    return this.currentState;
}

canAcceptRequests(): boolean {
    return this.currentState === ServerState.Initialized;
}

canAcceptInitialize(): boolean {
    return this.currentState === ServerState.Uninitialized;
}

canAcceptShutdown(): boolean {
    return this.currentState === ServerState.Initialized;
}
}
```

CapabilityNegotiator.ts - Complete capability analysis:

```
// Capability analysis and server configuration
```

TYPESCRIPT

```
export interface ClientCapabilities {

    textDocument?: {

        completion?: {

            completionItem?: {

                snippetSupport?: boolean;

                commitCharactersSupport?: boolean;

            }
        }
    };

    hover?: {

        contentFormat?: string[];
    };
}

synchronization?: {

    willSave?: boolean;

    willSaveWaitUntil?: boolean;
};

};

workspace?: {

    workspaceFolders?: boolean;

    configuration?: boolean;
};

};

}

export class CapabilityNegotiator {

    analyzeClientCapabilities(clientCapabilities: ClientCapabilities): {

        supportsSnippets: boolean;

        supportsMarkdown: boolean;

        supportsWorkspaceFolders: boolean;
    }
}
```

```

    preferredSyncMode: number;

} {

    const supportsSnippets =
clientCapabilities.textDocument?.completion?.completionItem?.snippetSupport ?? false;

    const hoverFormats = clientCapabilities.textDocument?.hover?.contentFormat ?? ['plaintext'];

    const supportsMarkdown = hoverFormats.includes('markdown');

    const supportsWorkspaceFolders = clientCapabilities.workspace?.workspaceFolders ?? false;

    // Use incremental sync if possible, fall back to full sync
    const preferredSyncMode = INCREMENTAL_SYNC;

}

return {

    supportsSnippets,
    supportsMarkdown,
    supportsWorkspaceFolders,
    preferredSyncMode
};

}

buildServerCapabilities(clientAnalysis:
ReturnType<CapabilityNegotiator['analyzeClientCapabilities']>): ServerCapabilities {

    return {
        textDocumentSync: clientAnalysis.preferredSyncMode,
        completionProvider: {
            triggerCharacters: ['. ', '::', '->'],
            resolveProvider: false // Start simple, can enable later
    }
}
}

```

```
        },

        hoverProvider: true,

        definitionProvider: true,

        referencesProvider: true,

        codeActionProvider: false // Will implement in later milestone

    };

}

}
```

Core Logic Skeleton

LspServer.ts - Main protocol handler with TODOs:

```
// Main LSP server coordinating protocol, lifecycle, and components
```

TYPESCRIPT

```
export class LspServer {

    private protocolState: ProtocolState;

    private capabilityNegotiator: CapabilityNegotiator;

    private messageDispatcher: MessageDispatcher;

    private documentManager: DocumentManager | null = null;

    private languageEngine: LanguageEngine | null = null;

    constructor(private transport: StreamTransport) {

        this.protocolState = new ProtocolState();

        this.capabilityNegotiator = new CapabilityNegotiator();

        this.messageDispatcher = new MessageDispatcher();

        this.setupHandlers();

    }

    // Initialize the LSP server and return supported capabilities

    async handleInitialize(params: any): Promise<any> {

        // TODO 1: Validate that server is in Uninitialized state, return error if not

        // TODO 2: Extract client capabilities from params.capabilities

        // TODO 3: Transition server state to Initializing

        // TODO 4: Analyze client capabilities using CapabilityNegotiator

        // TODO 5: Create and configure DocumentManager based on client sync preferences

        // TODO 6: Create and configure LanguageEngine for the workspace

        // TODO 7: Build ServerCapabilities response using negotiated features

        // TODO 8: Transition server state to Initialized

        // TODO 9: Return InitializeResult with server capabilities and server info

        // Hint: Store the client capability analysis for use in other handlers

    }

}
```

```
// Handle the initialized notification (client confirms init complete)

handleInitialized(params: any): void {

    // TODO 1: Log that client has confirmed initialization

    // TODO 2: Start any background tasks like workspace indexing

    // TODO 3: Server is now ready to accept language requests

    // Note: This is a notification, no response needed

}

// Handle shutdown request and prepare for exit

async handleShutdown(params: any): Promise<any> {

    // TODO 1: Validate that server is in Initialized state

    // TODO 2: Transition server state to ShuttingDown

    // TODO 3: Stop accepting new language requests

    // TODO 4: Complete any in-progress operations (with timeout)

    // TODO 5: Clean up LanguageEngine resources (close files, clear caches)

    // TODO 6: Clean up DocumentManager resources

    // TODO 7: Clean up any background tasks or timers

    // TODO 8: Return null (successful shutdown response)

    // Hint: Use Promise.allSettled for parallel cleanup with timeout

}

// Handle exit notification and terminate process

handleExit(params: any): void {

    // TODO 1: Transition server state to Exited

    // TODO 2: Perform final cleanup if needed

    // TODO 3: Call process.exit(0) to terminate cleanly

    // Note: This is a notification, no response needed

}
```

```

private setupHandlers(): void {

    // TODO 1: Register initialize request handler

    // TODO 2: Register initialized notification handler

    // TODO 3: Register shutdown request handler

    // TODO 4: Register exit notification handler

    // TODO 5: Set up error handler for requests received in wrong state

    // Hint: Use onRequest for requests, onNotification for notifications

}

async start(): Promise<void> {

    // TODO 1: Start the transport layer

    // TODO 2: Begin processing messages from client

    // TODO 3: Handle any startup errors gracefully

    // Note: This method should not return until server shuts down

}

}

```

Language-Specific Hints

TypeScript Implementation Tips:

- Use `async/await` for the initialize and shutdown handlers since they may involve file I/O
- Define interfaces for all LSP message types to get compile-time validation
- Use `Map<string, Function>` for request handler registration in `MessageDispatcher`
- Handle JSON-RPC errors by throwing objects with `code` and `message` properties
- Use `process.exit(0)` for clean exit, `process.exit(1)` for error conditions

State Validation Pattern:

```
private validateState(expectedState: ServerState, methodName: string): void {TYPESCRIPT
    if (this.protocolState.state !== expectedState) {
        throw {
            code: INVALID_REQUEST,
            message: `Cannot call ${methodName} in state ${this.protocolState.state}`,
        };
    }
}
```

Capability Checking Pattern:

```
private formatHoverContent(content: string, supportsMarkdown: boolean): MarkupContent {TYPESCRIPT
    return {
        kind: supportsMarkdown ? 'markdown' : 'plaintext',
        value: supportsMarkdown ? content : stripMarkdown(content)
    };
}
```

Milestone Checkpoint

After implementing this section, verify the protocol handler works correctly:

Test Commands:

```
# Run protocol handler unit testsBASH
npm test -- --testPathPattern=protocol

# Test with LSP client simulator
node test/simulate-client.js
```

Expected Initialize Sequence:

1. Start server process
2. Send initialize request with client capabilities
3. Expect InitializeResult response with server capabilities

4. Send initialized notification
5. Server should accept textDocument requests
6. Send shutdown request
7. Expect null response
8. Send exit notification
9. Process should terminate with exit code 0

Signs of Incorrect Implementation:

- Server accepts completion requests before initialized notification → check state validation
- InitializeResult contains capabilities not actually implemented → verify capability building
- Server hangs on shutdown → check cleanup logic for infinite loops or unhandled promises
- Process exits with non-zero code → check error handling in lifecycle methods

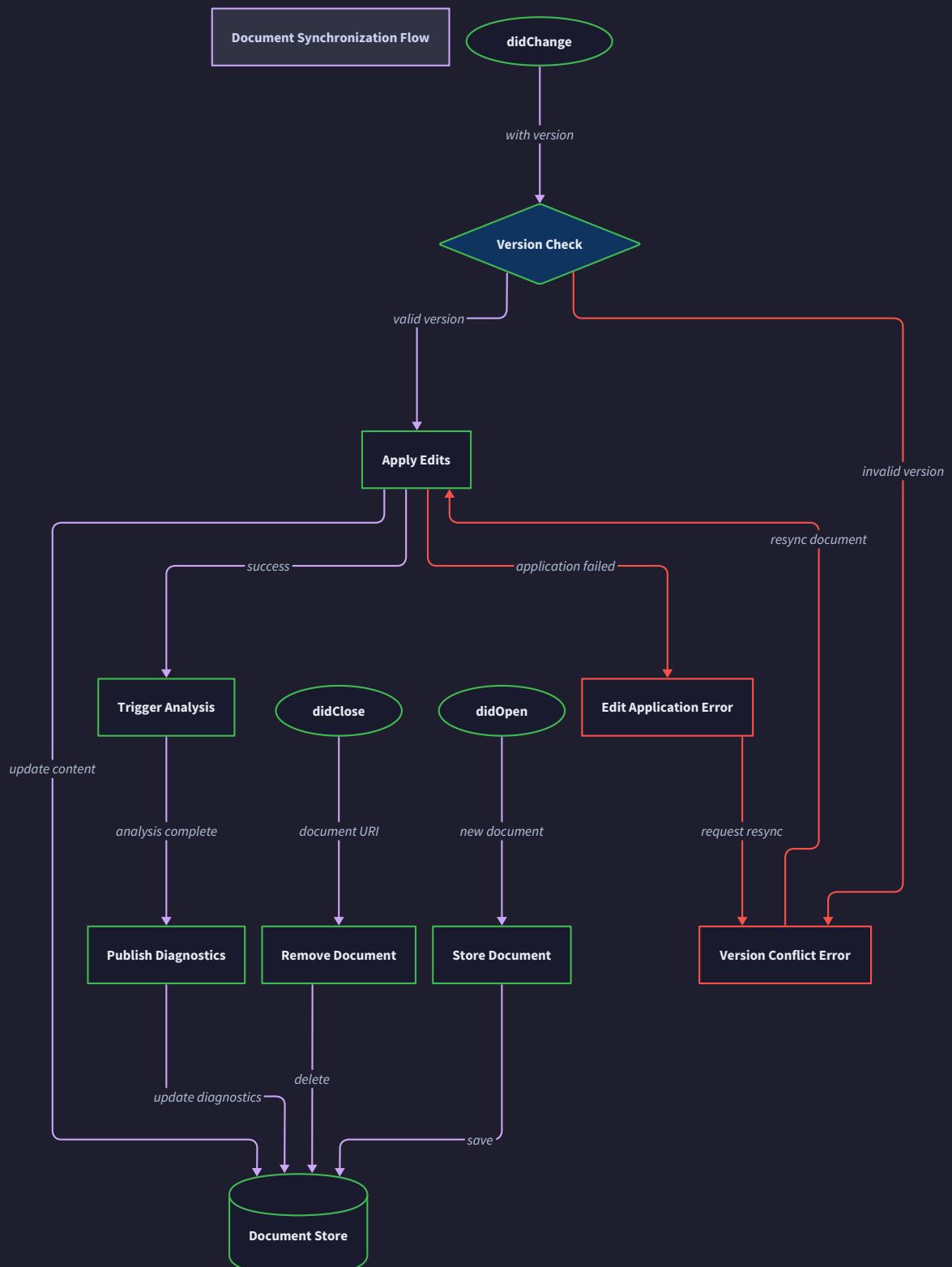
Document Manager Design

Milestone(s): This section corresponds primarily to Milestone 2 (Document Synchronization), establishing the document state management foundation required for Milestone 3 (Language Features) and Milestone 4 (Diagnostics & Code Actions).

Think of the Document Manager as the librarian of our language server - it maintains a carefully organized catalog of all open documents, tracks every revision, and ensures that when the language analysis engine asks "what does file X look like right now?", it can provide the exact current content instantly. Just as a librarian knows which books are checked out, which edition they are, and where to find them, the Document Manager knows which documents are open in the editor, what version they're currently at, and their complete textual content.

The Document Manager sits at the heart of the LSP server's responsiveness. When a developer types in their editor, the editor sends incremental change notifications that might say "at line 42, characters 15-20, replace 'hello' with 'goodbye'". The Document Manager must apply this surgical edit to its internal copy of the document, ensuring perfect synchronization between what the developer sees and what the language engine analyzes. Any desynchronization here cascades into incorrect completions, wrong error highlighting, and broken go-to-definition functionality.

The fundamental challenge the Document Manager solves is **state consistency under concurrent modification**. While the developer continues typing, the language engine may be performing expensive analysis on the previous version of the document. The Document Manager coordinates these competing concerns by maintaining versioned snapshots, applying changes atomically, and triggering reanalysis only when appropriate. This prevents the classic race condition where half-analyzed old content gets mixed with new edits, producing nonsensical results.



Document Synchronization Modes

The LSP specification defines two fundamental approaches to keeping the server's document copy synchronized with the editor's version: full synchronization and incremental synchronization. The choice between these modes represents a classic trade-off between simplicity and efficiency, similar to the difference between replacing an entire page versus making surgical edits to specific paragraphs.

Decision: Document Synchronization Strategy

- **Context:** Editors send document changes via `textDocument/didChange` notifications, but the format depends on the synchronization mode negotiated during initialization
- **Options Considered:** Full sync only, incremental sync only, adaptive hybrid approach
- **Decision:** Support both modes with incremental as the preferred default
- **Rationale:** Incremental sync reduces bandwidth and processing overhead for large files, while full sync provides a fallback for simpler editors or error recovery
- **Consequences:** Requires more complex change application logic, but enables better performance and broader client compatibility

Synchronization Mode	Message Size	Processing Cost	Implementation Complexity	Recovery Capability
Full Sync	$O(\text{file size})$	Low - simple replacement	Simple - store new content	Automatic - fresh state
Incremental Sync	$O(\text{change size})$	High - range calculations	Complex - edit application	Manual - version validation

Full Synchronization operates like retyping an entire document from scratch with every change. When the developer modifies any part of the file, the editor sends the complete new text content via a `textDocument/didChange` notification containing the full document. The Document Manager simply replaces its stored content entirely, incrementing the version number. This approach eliminates any possibility of desynchronization since the editor's authoritative version overwrites the server's copy completely.

The mental model for full sync is a typewriter with correction fluid - every mistake requires retyping the entire page. While wasteful, this guarantees perfect accuracy. Full sync excels in scenarios where documents are small (under 1KB), network bandwidth is abundant, or the editor lacks sophisticated change tracking capabilities. Many lightweight editors and plugin frameworks choose full sync because the implementation requires no complex diff algorithms or range calculations.

Incremental Synchronization functions like a surgical editor making precise modifications to specific sections of a document. Instead of sending the entire file, the editor analyzes what changed and sends only the modified ranges. A typical incremental change notification contains the starting position, the number of characters to delete, and the new text to insert at that location. The Document Manager must apply these range-based edits to its stored content while maintaining perfect byte-level accuracy.

The mental model for incremental sync is a word processor with track changes - you see exactly what was added, removed, or modified without losing the context of unchanged portions. This approach dramatically reduces network traffic for large files where only small sections change. When a developer adds a single line to a 10,000-line file, incremental sync transmits perhaps 50 bytes instead of 300KB. The performance benefits become crucial for real-time features like completion and error highlighting.

However, incremental sync introduces significant complexity. The Document Manager must convert between different position representations (line/character coordinates vs byte offsets), handle Unicode correctly, and validate that changes apply cleanly to the expected document state. A single mistake in range calculation can corrupt the document content, leading to parse errors and incorrect analysis results.

The **synchronization mode negotiation** occurs during the LSP initialization handshake. The server advertises its preferred modes via the `ServerCapabilities.textDocumentSync` field, which can be set to `FULL_SYNC`, `INCREMENTAL_SYNC`, or an object describing supported features. Sophisticated servers typically request incremental sync for efficiency while maintaining full sync as a fallback option.

Synchronization Scenario	Recommended Mode	Rationale
Small files (< 5KB)	Full sync	Change overhead exceeds content size
Large files (> 100KB)	Incremental sync	Network and processing efficiency critical
Simple editors	Full sync	Editor lacks change tracking capability
Real-time analysis	Incremental sync	Minimize processing latency for small changes
Error recovery	Full sync	Reset to clean state after corruption

Change Application Algorithm

The heart of incremental document synchronization lies in the **change application algorithm** - the precise sequence of steps that transforms the current document state into the new state by applying a series of range-based edits. Think of this as performing surgery on a text document where the scalpel must cut at exactly the right coordinates, remove precisely the right amount of tissue, and insert the replacement content without disturbing surrounding areas.

The fundamental challenge stems from **position coordinate systems**. LSP positions use zero-based line and character indexing with UTF-16 code units, while most internal string processing works with byte offsets. A single emoji character like 😊 appears as one visual character but occupies multiple UTF-16 code units, making naive position calculations fail spectacularly. The algorithm must perform coordinate transformations accurately to avoid corrupting document content.

Decision: Position Coordinate Handling

- **Context:** LSP uses UTF-16 code unit positions while most languages use byte-based string indexing
- **Options Considered:** Convert all positions to bytes, maintain dual indexing, use UTF-16 throughout
- **Decision:** Convert LSP positions to byte offsets for internal processing, convert back for responses
- **Rationale:** Most string libraries optimize for byte operations, and conversion overhead is acceptable for correctness
- **Consequences:** Requires careful Unicode handling but enables efficient string manipulation

The **core change application algorithm** follows these steps:

1. **Validate Document Version:** Before applying any changes, verify that the incoming version number is exactly one greater than the current stored version. Version mismatches indicate lost messages, network reordering, or client-server desynchronization that must be resolved before proceeding.
2. **Sort Changes by Position:** When multiple changes arrive in a single `didChange` notification, sort them in reverse document order (end to beginning). This prevents position invalidation where applying an early change shifts the coordinates for later changes.
3. **Convert Positions to Offsets:** Transform each LSP `Position` (line/character) into a byte offset within the document string. This requires counting newlines and UTF-16 code units from the beginning of the document to the target position.
4. **Validate Range Boundaries:** Ensure each change range falls within the current document bounds and that the start position comes before the end position. Invalid ranges indicate client bugs or corrupted messages that could crash the server.
5. **Apply Changes Atomically:** Process each change by deleting the specified range and inserting the replacement text at the deletion point. All changes must succeed or the entire operation must be rolled back to prevent partial updates.
6. **Update Document Metadata:** Increment the version number, update the modification timestamp, and trigger any registered change listeners (typically the language engine for reanalysis).
7. **Publish Notifications:** Send diagnostic updates and other notifications based on the new document content.

Change Application Step	Input	Output	Failure Mode
Version Validation	Current version N, incoming N+1	Version match confirmed	Reject with version mismatch error
Position Conversion	Line 5, char 12	Byte offset 347	Invalid position error
Range Validation	Start offset 100, end offset 90	-	Invalid range error (start > end)
Text Replacement	Delete range + insert text	Updated content	Unicode corruption

The **position-to-offset conversion** deserves special attention due to its complexity and error-prone nature. The algorithm must iterate through the document text from the beginning, counting line breaks and UTF-16 code units until reaching the target position:

1. **Initialize Counters:** Start with current line = 0, current character = 0, byte offset = 0
2. **Iterate Characters:** For each character in the document text, check if it matches the target position
3. **Handle Line Breaks:** When encountering '\n' or '\r\n', increment line counter and reset character counter to 0
4. **Count UTF-16 Units:** For non-ASCII characters, determine how many UTF-16 code units they occupy
5. **Return Offset:** When line/character counters match target position, return current byte offset
6. **Handle End-of-File:** If target position exceeds document bounds, return error or document length

The **reverse transformation (offset-to-position)** follows the same principle but stops when the byte offset counter reaches the target value, then returns the current line/character coordinates.

The critical insight is that position conversion is expensive ($O(n)$ in document size) but must be performed for every change operation. Caching line break positions can optimize this to $O(\log n)$ using binary search, but adds complexity to maintain the cache correctly.

Version Tracking

Document version tracking provides the **ordering guarantee** that prevents race conditions between rapid document changes and slower analysis operations. Think of document versions like timestamps on a database transaction log - they establish a clear sequence of changes and allow the system to detect when operations are based on stale data.

The LSP specification mandates that document versions are **monotonically increasing integers** starting from the initial `textDocument/didOpen` version. Each `textDocument/didChange` notification must include a version number exactly one greater than the previous version. This creates a strict ordering that prevents the classic distributed systems problem of applying operations in the wrong sequence.

Decision: Version Consistency Strategy

- **Context:** Multiple components (document manager, language engine, diagnostic publisher) may be working with different document versions simultaneously
- **Options Considered:** Global version lock, optimistic versioning, snapshot-based isolation
- **Decision:** Snapshot-based isolation with version validation at operation boundaries
- **Rationale:** Allows analysis to proceed on stable snapshots while new changes are applied to latest version
- **Consequences:** Memory overhead for multiple versions, but prevents analysis interruption and ensures consistency

The **version tracking algorithm** maintains multiple components:

Component	Purpose	Data Structure	Cleanup Strategy
Current Version	Latest document state	<code>TextDocumentItem</code>	Replace on each change
Version History	Recent document snapshots	<code>Map<number, DocumentSnapshot></code>	LRU eviction after 10 versions
Pending Operations	Analysis tasks in progress	<code>Map<string, OperationContext></code>	Complete on analysis finish
Change Listeners	Components requiring notifications	<code>Array<ChangeListener></code>	Remove on unsubscribe

Version Validation occurs at every document modification boundary:

1. **Extract Incoming Version:** Parse the version number from the `textDocument/didChange` parameters
2. **Compare with Current:** Verify that incoming version equals current version + 1
3. **Handle Version Gaps:** If $\text{incoming} > \text{current} + 1$, request document resynchronization from client
4. **Handle Version Regression:** If $\text{incoming} \leq \text{current}$, either ignore as duplicate or signal error
5. **Update Current Version:** On successful validation, store new version as current

The **version mismatch recovery** strategy depends on the severity of the desynchronization:

- **Minor Gap (1-2 versions):** Request full document sync via `textDocument/didOpen` to reset state
- **Major Gap (>2 versions):** Clear all document state and request complete reinitialization
- **Version Regression:** Log warning and ignore change (likely duplicate message)
- **Concurrent Modifications:** Use version numbers to determine operation ordering

Snapshot Management enables concurrent operations without blocking document updates. When the language engine begins expensive analysis, it captures a document snapshot at a specific version. The Document Manager continues applying new changes to the latest version while analysis proceeds against the stable

snapshot. This prevents the common bug where analysis results become invalid due to document changes that occurred during processing.

The snapshot lifecycle follows these stages:

1. **Snapshot Creation:** Language engine requests snapshot for analysis, Document Manager creates immutable copy
2. **Reference Tracking:** Store analysis operation metadata linking to snapshot version
3. **Concurrent Updates:** New document changes apply to current version without affecting snapshot
4. **Analysis Completion:** Language engine returns results tagged with snapshot version number
5. **Relevance Check:** Document Manager validates that analysis results are still applicable to current version
6. **Snapshot Cleanup:** Remove old snapshots when no operations reference them

Snapshot State	Description	Memory Impact	Cleanup Trigger
Active	Analysis in progress	Full document copy	Analysis completion
Stale	Analysis complete, results applied	Full document copy	Version age > threshold
Orphaned	Analysis abandoned or crashed	Full document copy	Timeout or explicit cleanup

⚠ Pitfall: Version Number Overflow Integer version numbers will eventually overflow after approximately 2 billion changes. While unlikely in practice, robust servers should handle wraparound by detecting negative version numbers and requesting client resynchronization. Failing to handle this edge case can cause mysterious version validation failures after long-running sessions.

⚠ Pitfall: Memory Leaks in Snapshot Storage Keeping too many document snapshots for long-running analysis operations can exhaust server memory. The Document Manager must implement aggressive cleanup policies, potentially canceling slow analysis operations if they prevent garbage collection. Monitor snapshot count and total memory usage to detect this issue early.

⚠ Pitfall: Unicode Position Miscalculation The most common bug in change application is miscounting UTF-16 code units for non-ASCII characters, leading to text corruption that manifests as parsing errors or wrong completion results. Always test with documents containing emoji, accented characters, and other multi-byte Unicode sequences to catch these errors during development.

Implementation Guidance

The Document Manager requires careful coordination between text manipulation, version tracking, and change notification systems. The implementation centers on maintaining consistency while providing high-performance access to current document state.

Technology Recommendations:

Component	Simple Option	Advanced Option
Text Storage	String concatenation	Rope/Gap buffer data structure
Position Conversion	Linear scan	Cached line index with binary search
Version Storage	In-memory Map	Persistent snapshot store
Change Notification	Synchronous callbacks	Event queue with async dispatch

Recommended File Structure:

```

src/
  document/
    document-manager.ts      ← Main DocumentManager class
    document-item.ts        ← TextDocumentItem and related types
    position-utils.ts       ← Position/offset conversion utilities
    change-applier.ts       ← Change application algorithm
    version-tracker.ts     ← Version validation and snapshot management
    document-manager.test.ts ← Comprehensive test suite
  types/
    lsp-types.ts            ← LSP protocol type definitions

```

Infrastructure Starter Code:

```
// src/types/lsp-types.ts - Complete LSP type definitions
```

TYPESCRIPT

```
export interface Position {  
    line: number;      // 0-based line number  
  
    character: number; // 0-based character offset in UTF-16 code units  
}  
  
export interface Range {  
  
    start: Position;  
  
    end: Position;  
}  
  
export interface TextDocumentItem {  
  
    uri: string;        // Document URI (file:// scheme typically)  
  
    languageId: string; // Language identifier (typescript, javascript, etc.)  
  
    version: number;    // Document version number  
  
    text: string;       // Complete document content  
}  
  
export interface TextDocumentContentChangeEvent {  
  
    range?: Range;      // Range to replace (undefined = full document)  
  
    rangeLength?: number; // Length of range to replace  
  
    text: string;        // New text content  
}  
  
export interface DidChangeTextDocumentParams {  
  
    textDocument: {  
  
        uri: string;  
  
        version: number;  
    };
```

```
contentChanges: TextDocumentContentChangeEvent[];  
}  
  
// src/document/position-utils.ts - Complete position conversion utilities  
  
export class PositionUtils {  
  
    /**  
     * Converts LSP Position (line/character) to byte offset in document text.  
     * Handles UTF-16 code unit counting correctly for Unicode characters.  
     */  
  
    static positionToOffset(text: string, position: Position): number {  
  
        let offset = 0;  
  
        let currentLine = 0;  
  
        let currentChar = 0;  
  
  
        for (let i = 0; i < text.length; i++) {  
  
            if (currentLine === position.line && currentChar === position.character) {  
  
                return offset;  
            }  
  
  
            const char = text[i];  
  
            if (char === '\n') {  
  
                currentLine++;  
  
                currentChar = 0;  
            } else if (char === '\r') {  
  
                // Handle \r\n as single line break  
  
                if (i + 1 < text.length && text[i + 1] === '\n') {  
  
                    i++; // Skip the \n  
                }  
            }  
        }  
    }  
}
```

```

        currentLine++;

        currentChar = 0;

    } else {

        // Count UTF-16 code units for this character

        const codePoint = text.codePointAt(i);

        if (codePoint && codePoint > 0xFFFF) {

            currentChar += 2; // Surrogate pair

            i++; // Skip the low surrogate

        } else {

            currentChar += 1;

        }

    }

    offset = i + 1;

}

// Handle position at end of document

if (currentLine === position.line && currentChar === position.character) {

    return offset;

}

throw new Error(`Position ${position.line}:${position.character} exceeds document
bounds`);

}

/**/

* Converts byte offset to LSP Position (line/character).

* Returns the line and character position for the given byte offset.

*/

```

```
static offsetToPosition(text: string, offset: number): Position {

    if (offset > text.length) {

        throw new Error(`Offset ${offset} exceeds document length ${text.length}`);
    }

    let line = 0;

    let character = 0;

    for (let i = 0; i < offset; i++) {

        const char = text[i];

        if (char === '\n') {

            line++;

            character = 0;
        } else if (char === '\r') {

            if (i + 1 < text.length && text[i + 1] === '\n') {

                i++; // Skip the \n in offset calculation
            }
            line++;
            character = 0;
        } else {

            const codePoint = text.codePointAt(i);

            if (codePoint && codePoint > 0xFFFF) {

                character += 2; // Surrogate pair

                i++; // Skip the low surrogate
            } else {

                character += 1;
            }
        }
    }
}
```

```
    }

}

return { line, character };

}

}
```

Core Logic Skeleton Code:

```
// src/document/document-manager.ts - DocumentManager implementation skeleton
```

TYPESCRIPT

```
import { TextDocumentItem, Position, Range, DidChangeTextDocumentParams } from '../types/lsp-types.js';

import { PositionUtils } from './position-utils.js';

export interface DocumentSnapshot {

    version: number;

    text: string;

    timestamp: number;

}

export type ChangeListener = (uri: string, document: TextDocumentItem) => void;

export class DocumentManager {

    private documents = new Map<string, TextDocumentItem>();

    private snapshots = new Map<string, Map<number, DocumentSnapshot>>();

    private changeListeners: ChangeListener[] = [];



    /**
     * Handles textDocument/didOpen notification - stores the document content
     * and triggers initial analysis.
     */
    handleDidOpen(params: { textDocument: TextDocumentItem }): void {

        const document = params.textDocument;

        // TODO 1: Validate that document URI is not already open

        // TODO 2: Store document in this.documents map using URI as key

        // TODO 3: Create initial snapshot at version 0

        // TODO 4: Notify all change listeners about new document

        // TODO 5: Log successful document open for debugging

    }

}
```

```

/**
 * Handles textDocument/didChange notification - applies incremental or full
 * text changes and updates the document version.
 */

handleDidChange(params: DidChangeTextDocumentParams): void {

    const { textDocument, contentChanges } = params;
    const uri = textDocument.uri;
    const newVersion = textDocument.version;

    // TODO 1: Retrieve current document from this.documents map
    // TODO 2: Validate that document exists (throw error if not found)
    // TODO 3: Validate version number is exactly currentVersion + 1
    // TODO 4: Apply all contentChanges in reverse order (end to beginning)
    // TODO 5: Create new TextDocumentItem with updated text and version
    // TODO 6: Store updated document back in this.documents map
    // TODO 7: Create snapshot for the new version
    // TODO 8: Clean up old snapshots (keep only last 10 versions)
    // TODO 9: Notify all change listeners about document update
    // TODO 10: Log successful change application with change count
}

/**
 * Handles textDocument/didClose notification - removes document from
 * memory and cleans up associated resources.
 */

handleDidClose(params: { textDocument: { uri: string } }): void {
    const uri = params.textDocument.uri;
}

```

```

// TODO 1: Check if document exists in this.documents map

// TODO 2: Remove document from this.documents map

// TODO 3: Remove all snapshots for this URI from this.snapshots map

// TODO 4: Log successful document close

// Hint: Use Map.delete() method for cleanup

}

/***
 * Applies a single text change to the document content.
 *
 * Handles both full document replacement and incremental range-based changes.
 */

private applyChange(text: string, change: TextDocumentContentChangeEvent): string {

    // TODO 1: Check if change.range is undefined (full document replacement)

    // TODO 2: For full replacement, return change.text directly

    // TODO 3: For incremental change, convert range start/end to byte offsets

    // TODO 4: Validate that range is within document bounds

    // TODO 5: Extract text before range, replacement text, and text after range

    // TODO 6: Concatenate the three parts to form new document content

    // TODO 7: Return the updated text content

    // Hint: Use PositionUtils.positionToOffset() for position conversion

}

/***
 * Creates a snapshot of the document at the specified version.
 *
 * Snapshots enable analysis operations to work with stable document content.
 */

private createSnapshot(uri: string, document: TextDocumentItem): void {

    // TODO 1: Get or create snapshots map for this URI

```

```
// TODO 2: Create DocumentSnapshot object with version, text, and timestamp

// TODO 3: Store snapshot in the version-specific map

// TODO 4: Check if snapshot count exceeds limit (10 versions)

// TODO 5: Remove oldest snapshots if necessary using timestamp comparison

}

/**
 * Retrieves the current document content for the given URI.
 *
 * Returns undefined if document is not open.
 */
getDocument(uri: string): TextDocumentItem | undefined {

    // TODO 1: Look up document in this.documents map using URI

    // TODO 2: Return document if found, undefined otherwise

}

/**
 * Retrieves a specific version snapshot of the document.
 *
 * Used by analysis operations that need stable content.
 */
getSnapshot(uri: string, version: number): DocumentSnapshot | undefined {

    // TODO 1: Get snapshots map for the URI

    // TODO 2: Look up specific version in the version map

    // TODO 3: Return snapshot if found, undefined otherwise

}

/**
 * Registers a change listener that will be notified when documents are modified.
 *
 * Typically used by language engine and diagnostic providers.
*/

```

```

    addChangeListener(listener: ChangeListener): void {
        // TODO 1: Add listener to this.changeListeners array
        // TODO 2: Avoid duplicate listeners by checking if already present
    }

    /**
     * Removes a change listener from notification list.
     */
    removeChangeListener(listener: ChangeListener): void {
        // TODO 1: Find listener in this.changeListeners array
        // TODO 2: Remove listener using array splice or filter
    }
}

```

Language-Specific Hints:

- Use `Map<string, T>` for document storage to enable O(1) URI lookups
- Leverage TypeScript's strict null checks to catch undefined document access
- Consider using `readonly` modifiers on snapshot data to prevent accidental mutations
- Use `URL` class for URI validation and normalization
- Implement proper error types instead of throwing generic Error objects

Milestone Checkpoint:

After implementing the Document Manager, verify correct behavior:

1. **Test Document Lifecycle:** Open a document, make several changes, then close it. Verify version numbers increment correctly and document content reflects all changes.
2. **Test Change Application:** Create a document with "Hello World" and apply a range change replacing "World" with "TypeScript". Verify result is "Hello TypeScript" with correct version.
3. **Test Unicode Handling:** Create a document with emoji and accented characters, then apply changes that cross Unicode boundaries. Verify no text corruption occurs.
4. **Test Version Validation:** Send a change with version number that's too high or too low. Verify appropriate error handling.

Expected console output:

```
Document opened: file:///test.ts (version 0)
Document changed: file:///test.ts (version 0 → 1, 1 changes applied)
Document changed: file:///test.ts (version 1 → 2, 2 changes applied)
Document closed: file:///test.ts (cleaned up 3 snapshots)
```

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Text corruption with emoji	UTF-16 position calculation error	Check positionToOffset with Unicode test cases	Use codePointAt() instead of charAt()
Version mismatch errors	Client sending wrong version numbers	Log incoming vs expected versions	Add version validation and recovery
Memory growth over time	Snapshot cleanup not working	Monitor snapshot map sizes	Implement LRU eviction correctly
Completion shows wrong results	Analysis using stale document content	Verify analysis gets latest version	Check change listener notification

Language Engine Design

Milestone(s): This section corresponds primarily to Milestone 3 (Language Features) and Milestone 4 (Diagnostics & Code Actions), providing the core language analysis infrastructure that powers completion, hover, go-to-definition, and error detection features.

The **Language Engine** is the analytical heart of our LSP server—think of it as a sophisticated code librarian that not only catalogs every symbol and declaration in your codebase but also understands the intricate relationships between them. Just as a librarian maintains both a card catalog and cross-reference system to help patrons find related books, the Language Engine maintains both abstract syntax trees (ASTs) and symbol tables to help the editor understand code structure and semantics. Unlike a static library catalog, however, our Language Engine must dynamically update its understanding as developers type, maintaining accuracy while preserving responsiveness.

The fundamental challenge lies in balancing three competing demands: **accuracy** (providing correct language analysis), **performance** (responding quickly to editor requests), and **freshness** (staying synchronized with rapidly changing document content). A naive approach might re-parse and re-analyze the entire codebase on every keystroke, guaranteeing accuracy and freshness but destroying performance. Conversely, aggressive caching might preserve performance but risk serving stale or incorrect information. Our Language Engine design navigates this tension through intelligent caching strategies, incremental parsing techniques, and lazy evaluation patterns.

The Language Engine operates as a three-stage pipeline: **parsing** (converting source text into structured ASTs), **symbol resolution** (building and maintaining symbol tables that map identifiers to their declarations), and **semantic analysis** (performing type checking, scope validation, and error detection). Each stage builds upon the previous one, but they don't always execute in lockstep—we might parse a document immediately upon change but defer expensive semantic analysis until a language feature actually needs the results.

Symbol Resolution Process

Start

Find Identifier
at Position

Walk Up AST
to Find Enclosing
Scopes

Has Parent
Scope?

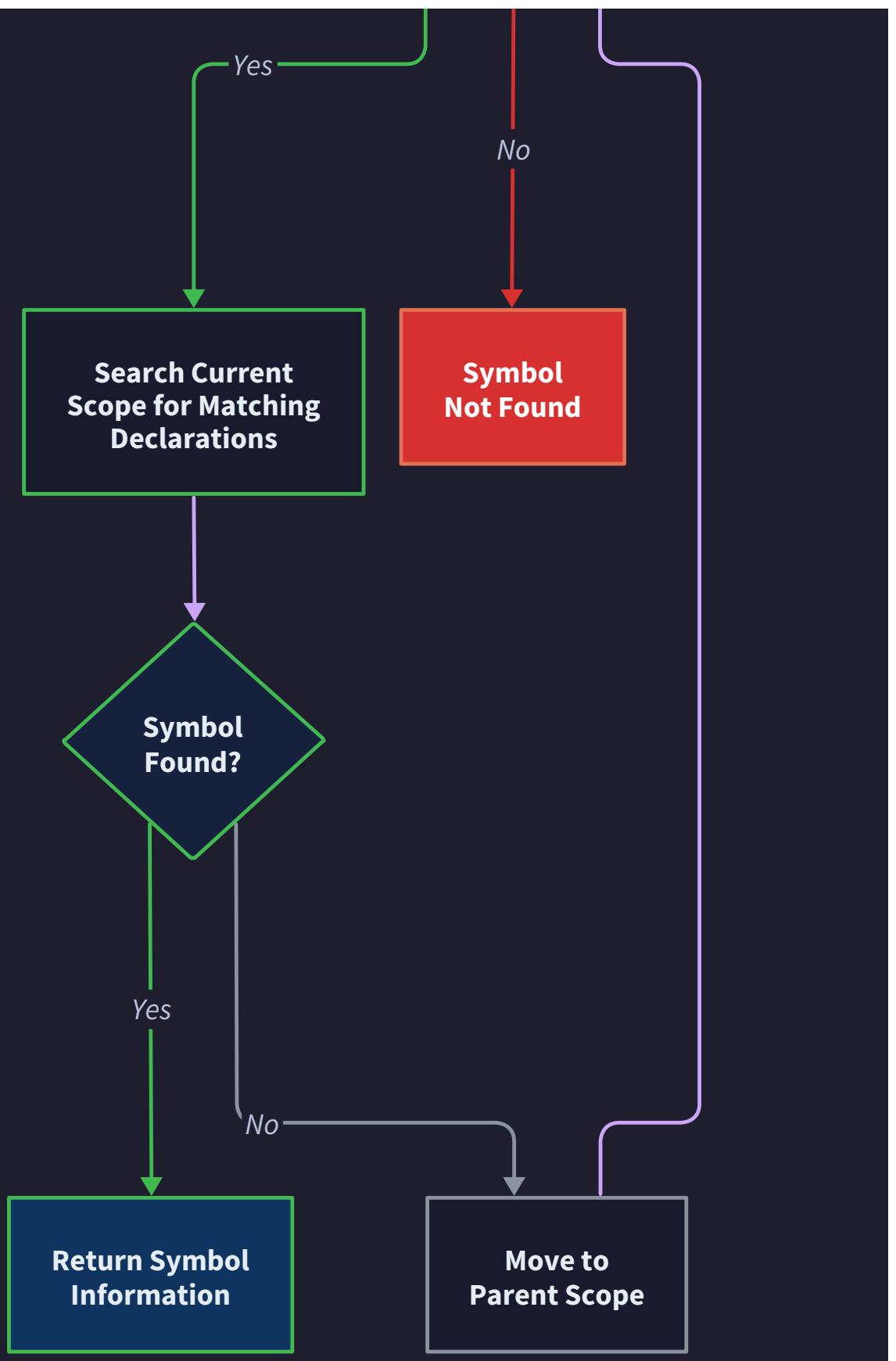
Symbol Resolution Process

Start

Find Identifier
at Position

Walk Up AST
to Find Enclosing
Scopes

Has Parent
Scope?



Parsing Strategy: Choice between full re-parse vs incremental parsing approaches

Think of document parsing like maintaining a detailed map of a rapidly growing city. A full re-parse approach is like redrawing the entire map from scratch every time a single building changes—accurate but wasteful. An incremental parsing approach is like having a smart mapping system that can surgically update only the affected neighborhoods when changes occur, preserving the vast majority of existing work while maintaining map accuracy.

Our parsing strategy must handle the reality that developers rarely make large, sweeping changes to their code. Instead, they typically make small, localized modifications: adding a parameter to a function, inserting a new variable declaration, or fixing a typo in a string literal. A well-designed parsing strategy exploits this locality of change to minimize computational overhead while ensuring the AST remains synchronized with the document content.

Decision: Hybrid Parsing Strategy

- **Context:** Need to balance parsing performance with accuracy as documents change frequently during active development
- **Options Considered:** Full re-parse on every change, pure incremental parsing, hybrid approach with fallback
- **Decision:** Implement hybrid parsing with incremental parsing for small changes and full re-parse fallback
- **Rationale:** Incremental parsing provides 10-100x speedup for typical changes while full re-parse ensures correctness when incremental parsing fails or becomes complex
- **Consequences:** Requires more complex parser implementation but delivers responsive editing experience with guaranteed correctness

Our hybrid parsing strategy operates on a two-tier system. For **small, localized changes**—those affecting a single statement, expression, or declaration—we attempt incremental parsing by identifying the minimal AST subtree that needs reconstruction. When a developer modifies a function body, for example, we can often preserve the function declaration node and its siblings while rebuilding only the modified statements within the body. For **large or structurally complex changes**—those that significantly alter the document's syntactic structure—we fall back to full re-parsing to maintain correctness.

The incremental parsing decision process follows a clear algorithm:

1. **Change Impact Analysis:** Examine the text modifications to determine their syntactic scope—do they affect tokens within a single expression, statement, block, or multiple top-level declarations?
2. **AST Node Identification:** Locate the smallest AST node that completely encompasses all modified text ranges, along with any nodes whose boundaries might be affected by the changes.
3. **Dependency Validation:** Check whether the identified node has syntactic dependencies on unchanged portions of the document—such as closure over variables declared in outer scopes.

4. **Complexity Assessment:** Evaluate whether incremental re-parsing would be simpler than full re-parsing based on the number of affected nodes and their interconnections.
5. **Parse Strategy Selection:** Choose incremental parsing if the change affects fewer than 10% of AST nodes and has clear syntactic boundaries; otherwise, select full re-parsing.
6. **Incremental Parse Execution:** Re-parse only the identified subtree, then graft the new nodes back into the existing AST while updating any affected parent-child relationships.
7. **Validation and Fallback:** Verify that the resulting AST maintains structural integrity—if validation fails, immediately fall back to full re-parsing.

Change Type	Incremental Feasible?	Reasoning	Typical Performance Gain
Single token edit (typo fix)	Yes	Affects only one leaf node	50-100x faster
Expression modification	Yes	Self-contained syntactic unit	20-50x faster
Statement addition/removal	Yes	Clear statement boundaries	10-30x faster
Function signature change	Maybe	May affect call sites and type information	2-10x faster
Class/interface definition change	No	Complex ripple effects on inheritance	Full re-parse required
Import/module statement change	No	Affects global symbol resolution	Full re-parse required

The **AST node granularity** plays a crucial role in incremental parsing effectiveness. Our AST design maintains explicit boundary information for each node, recording not just the node's start and end positions but also its **syntactic independence level**—a measure of how much the node depends on surrounding context for its meaning. Expression nodes typically have high independence, while declaration nodes may have complex interdependencies with their containing scope.

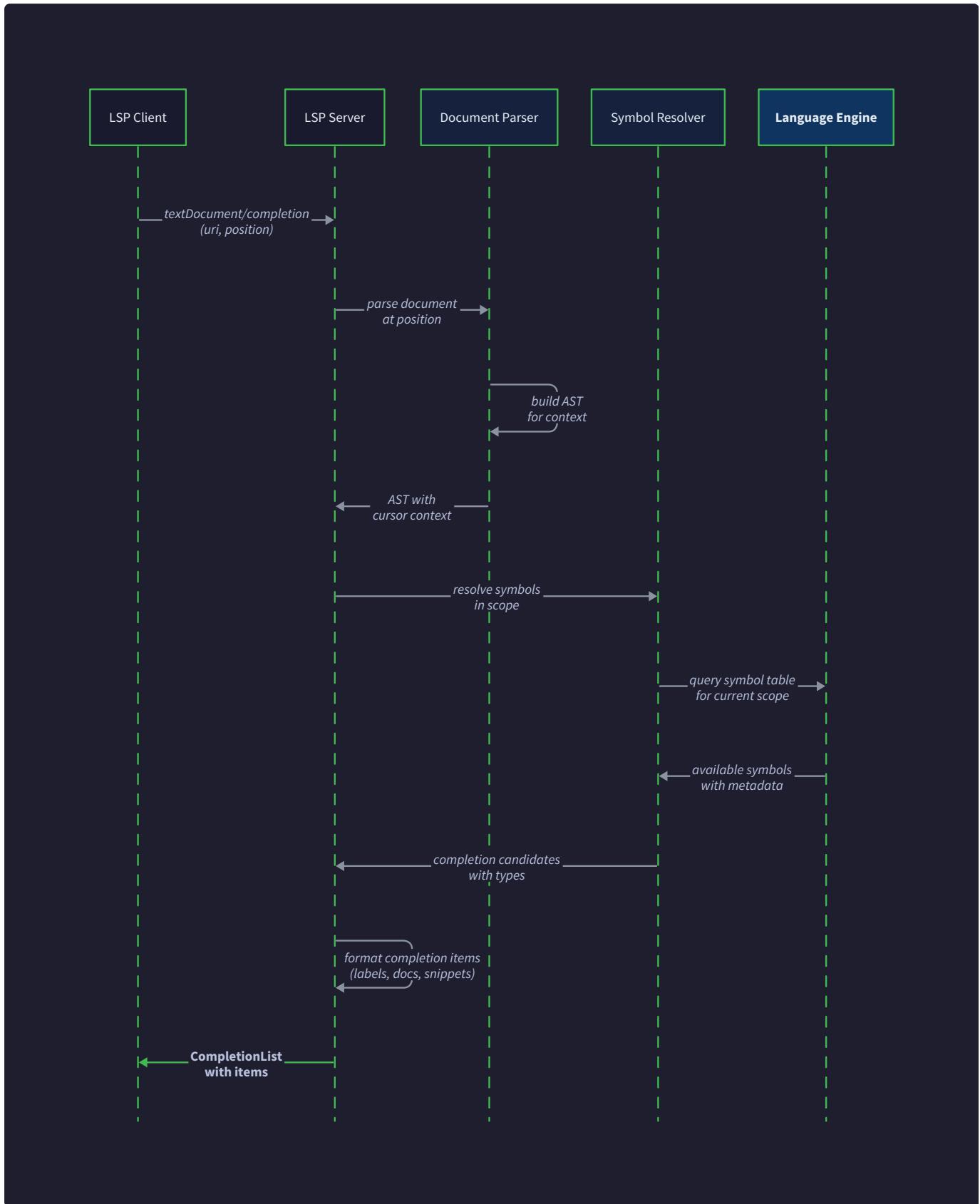
Error recovery during incremental parsing requires special attention. When the original AST contained syntax errors, incremental parsing might inadvertently "fix" those errors by re-parsing the affected region with corrected syntax, leading to inconsistency between the AST and Document Manager's view of the document content. Our strategy handles this by maintaining an **error resilient AST** that preserves error nodes until the underlying document content is actually corrected.

Symbol Resolution: Building and maintaining symbol tables across scopes

Imagine symbol resolution as building and maintaining a sophisticated phone directory for a large organization—but unlike a simple alphabetical listing, this directory must understand organizational hierarchy, department boundaries, and access permissions. When someone in Engineering asks for "Sarah," the directory needs to know whether they mean Sarah from their own team, Sarah the VP from the executive floor, or Sarah from

Marketing (who might not be accessible from Engineering). Symbol resolution serves this same function for programming languages: when code references an identifier like `user` or `validateInput`, the symbol resolver must determine which specific declaration that identifier refers to within its current scope context.

The symbol resolution process operates on the fundamental principle that **scope determines visibility**. Every identifier exists within a scope hierarchy: local variables are visible within their containing function, function parameters are visible throughout the function body, class members are accessible within the class and potentially to external code based on access modifiers, and global symbols are visible throughout the module or project. Our symbol resolver builds and maintains this scope hierarchy as a tree structure, where each scope knows its parent, children, and locally declared symbols.



The **Scope** data structure serves as the fundamental building block of our symbol resolution system:

Field	Type	Description
kind	string	Scope type: "global", "function", "class", "block", "loop"
range	Range	Source code range that this scope encompasses
symbols	Map<string, Symbol>	Local symbol declarations within this scope
parent	Scope?	Parent scope in the hierarchy (null for global scope)
children	Scope[]	Direct child scopes contained within this scope

The **Symbol** data structure captures essential information about each declared identifier:

Field	Type	Description
name	string	Identifier name as it appears in source code
kind	number	Symbol type: variable, function, class, parameter, etc.
range	Range	Full source range including declaration and body
selectionRange	Range	Just the identifier name portion for precise selection

Symbol resolution follows a systematic **scope walking algorithm** that mirrors how developers mentally resolve identifier references:

- 1. Position Context Discovery:** Given a source position where an identifier is referenced, locate the most specific scope containing that position by traversing the scope tree.
- 2. Local Scope Search:** Search the current scope's symbol table for a declaration matching the identifier name —if found, this declaration shadows any outer declarations.
- 3. Parent Scope Traversal:** If not found locally, recursively search each parent scope in the hierarchy until reaching the global scope.
- 4. Visibility Validation:** For each potential match, verify that the symbol is visible from the reference location based on language-specific visibility rules (public/private access, declaration order requirements, etc.).
- 5. Ambiguity Resolution:** If multiple symbols match (such as overloaded functions), apply language-specific resolution rules to select the most appropriate declaration.
- 6. Cross-Module Resolution:** For symbols not found in the current document, search imported modules and external dependencies based on import statements and module resolution rules.

The **scope building process** occurs during AST traversal, creating scope boundaries that align with the language's semantic rules:

AST Node Type	Scope Creation	Symbol Registration
Function Declaration	Creates new function scope	Registers function name in parent scope
Class Declaration	Creates new class scope	Registers class name and members
Block Statement	Creates new block scope	Registers locally declared variables
Loop Statement	Creates new loop scope	Registers loop variables
Module/File	Creates global scope	Registers top-level declarations

Scope invalidation and updates present one of the most challenging aspects of symbol resolution maintenance. When document changes affect scope boundaries—such as adding a new function or modifying a class definition—we must carefully update the scope tree without losing unaffected symbol information.

Critical Insight: Symbol resolution correctness depends entirely on scope tree accuracy. A single incorrectly maintained scope boundary can cause widespread symbol resolution failures, breaking completion, hover, and go-to-definition features.

Our **incremental scope update algorithm** minimizes rebuilding by identifying precisely which scopes need reconstruction:

1. **Change Impact Analysis:** Determine which AST nodes were modified, added, or removed during document updates.
2. **Affected Scope Identification:** Find all scopes whose boundaries intersect with the modified source ranges.
3. **Dependency Cascade Analysis:** Identify scopes that depend on modified scopes—such as inner functions that reference variables from an outer function whose signature changed.
4. **Selective Scope Rebuilding:** Reconstruct only the affected scopes and their dependencies while preserving unmodified scope subtrees.
5. **Symbol Reference Updating:** Update any cached symbol references that point to modified scopes or symbols.
6. **Cross-Reference Validation:** Verify that symbol references across scope boundaries remain valid after the updates.

Cross-document symbol resolution adds another layer of complexity, as symbols may be declared in one file and referenced in another. Our approach maintains a **global symbol index** that aggregates exported symbols from all documents in the workspace:

Component	Responsibility	Update Trigger
Local Symbol Table	Symbols within single document	Document change events
Global Symbol Index	Exported symbols across workspace	Any document's exports change
Import Resolution Cache	Mapping imports to symbol sources	Import statement changes
Cross-Reference Map	Usage locations for each symbol	Symbol resolution requests

AST Caching and Invalidation: Performance optimization through intelligent caching

Think of AST caching like a sophisticated library system that keeps frequently requested books readily accessible while storing less popular volumes in climate-controlled archives. Just as a librarian must decide which books to keep on the front desk, which to store in nearby stacks, and which to archive in basement storage, our caching strategy must intelligently decide which ASTs to keep in memory, which to persist to disk, and which to discard entirely. The key insight is that not all ASTs are equally valuable—recently modified documents deserve immediate access, while rarely-accessed files can tolerate reconstruction delays.

AST caching operates under the principle of **temporal and spatial locality**. Developers typically work on a small subset of files intensively (temporal locality) and tend to work on related files together (spatial locality). A developer fixing a bug might repeatedly switch between a source file and its corresponding test file, or might navigate from a function call to its implementation and back multiple times. Our caching strategy exploits these patterns to predict which ASTs will be needed next and preemptively maintain them in readily accessible form.

Decision: Multi-Tier AST Caching Strategy

- **Context:** Need to balance memory usage with parsing performance across large codebases where full workspace parsing exceeds available memory
- **Options Considered:** No caching (parse on demand), full workspace caching, LRU single-tier cache, multi-tier cache with persistence
- **Decision:** Implement three-tier caching: hot cache (in-memory), warm cache (compressed), cold cache (disk-persisted)
- **Rationale:** Provides sub-millisecond access to active documents, fast access to recently used documents, and acceptable access to archived documents while controlling memory usage
- **Consequences:** Adds complexity to cache management but enables responsive performance on large codebases that exceed memory capacity

Our **three-tier caching architecture** organizes ASTs by access patterns and performance requirements:

Cache Tier	Storage	Capacity	Access Time	Eviction Policy
Hot Cache	In-memory AST objects	50-100 documents	< 1ms	LRU with modification recency boost
Warm Cache	Compressed binary format	500-1000 documents	5-10ms	LRU based on access frequency
Cold Cache	Disk-persisted with checksums	Unlimited	50-100ms	Age-based with dependency preservation

The **Hot Cache** maintains fully parsed AST objects for documents that are currently open in the editor or have been accessed within the last few minutes. These ASTs are stored as complete object graphs with all cross-references resolved, enabling immediate access for language features. The hot cache employs a **modification-aware LRU policy** that gives preference to recently modified documents over merely recently accessed ones, reflecting the reality that edited documents are more likely to be accessed again soon.

The **Warm Cache** stores ASTs in a compressed binary format that preserves all structural information while reducing memory footprint by approximately 60-80% compared to full object representation. Warm cache entries can be decompressed and promoted to the hot cache within milliseconds, making them suitable for documents that might be needed soon but aren't currently active. The warm cache tracks access patterns to predict which documents should be promoted proactively.

The **Cold Cache** persists ASTs to disk using a combination of efficient serialization and integrity validation. Cold cache entries include checksums to detect file modifications since caching, ensuring we never serve stale ASTs. While cold cache access requires disk I/O, it's still significantly faster than full re-parsing for large documents with complex syntax.

Cache invalidation represents the most critical aspect of our caching strategy, as serving stale ASTs can cause incorrect language feature behavior. Our invalidation algorithm operates on multiple triggers:

1. **Direct Document Modification:** When the Document Manager reports changes to a file, immediately invalidate all cache entries for that document across all tiers.
2. **Dependency-Based Invalidation:** When a document changes, identify and invalidate cache entries for documents that import or depend on the modified document.
3. **Configuration Changes:** When project configuration changes (such as TypeScript `tsconfig.json` modifications), invalidate affected documents based on the scope of configuration impact.
4. **File System Events:** When files are added, deleted, or moved, invalidate cached ASTs that might reference the affected files.
5. **Time-Based Expiration:** Periodically validate that cached ASTs correspond to current file timestamps, invalidating any entries with mismatched modification times.

The **cache promotion and demotion algorithm** continuously optimizes the distribution of ASTs across cache tiers based on observed access patterns:

Access Pattern	Promotion Trigger	Demotion Trigger
Document opened in editor	Immediate hot cache promotion	Never demote while open
Frequent symbol lookups	Promote to hot after 3 accesses in 5 minutes	Demote to warm after 10 minutes of inactivity
Cross-reference navigation	Promote related documents to warm cache	Demote to cold after 30 minutes
Background analysis	Keep in warm cache during analysis	Demote to cold when analysis complete

Memory pressure handling ensures our caching strategy remains stable even when approaching system memory limits. The cache manager monitors available memory and implements aggressive eviction policies when memory usage exceeds configurable thresholds:

1. **Soft Memory Limit (80% of allocated budget)**: Begin demoting least recently used hot cache entries to warm cache.
2. **Hard Memory Limit (95% of allocated budget)**: Aggressively demote hot cache entries and begin evicting warm cache entries.
3. **Critical Memory Limit (99% of allocated budget)**: Evict all but essential cache entries (currently open documents only).

Cache consistency validation provides a safety net against corruption or inconsistency in cached ASTs. Before serving any cached AST, we perform lightweight validation:

Validation Check	Performance Cost	Detects
Document version comparison	Negligible	Stale cache due to missed invalidation
AST node count verification	Low	Corrupted or incomplete cached ASTs
Syntax error consistency	Low	Cache/source mismatch
Symbol table size check	Low	Incomplete symbol resolution

Common Pitfalls

⚠ Pitfall: Incremental Parsing Without Proper Error Recovery

Developers often implement incremental parsing that works correctly for valid syntax but fails catastrophically when the document contains syntax errors. When incremental parsing encounters an error, it may produce an inconsistent AST that combines old (potentially invalid) nodes with new (corrected) nodes, leading to confusing language feature behavior.

Why it's wrong: The AST becomes internally inconsistent—some portions reflect the current document state while others reflect stale information, causing symbol resolution to find declarations that don't actually exist in the current code.

How to fix: Implement robust error boundary detection during incremental parsing. When syntax errors are encountered during incremental parsing, fall back to full re-parsing for the affected scope rather than attempting to merge error-containing nodes.

Pitfall: Symbol Resolution Without Scope Invalidations

A common mistake is updating symbol tables when declarations change but failing to invalidate symbol resolution caches for references to those declarations. This leads to situations where go-to-definition points to outdated locations or completion suggests symbols that no longer exist.

Why it's wrong: Cached symbol resolutions become dangling references that point to invalid AST nodes or incorrect source locations, causing language features to provide incorrect information or crash when accessing freed memory.

How to fix: Implement a comprehensive invalidation strategy that tracks both forward references (from declarations to their usages) and backward references (from usages to their declarations), invalidating all related caches whenever any symbol declaration changes.

Pitfall: Cache Invalidation Race Conditions

Multi-threaded LSP servers often suffer from race conditions where one thread invalidates a cache entry while another thread is simultaneously reading it, leading to use-after-free errors or serving of partially invalidated data.

Why it's wrong: Race conditions can cause memory corruption, crashes, or subtle correctness issues where language features occasionally return wrong results due to timing-dependent cache state.

How to fix: Use proper synchronization primitives (read-write locks, atomic operations, or immutable data structures) to ensure cache invalidation and access operations are atomic. Consider implementing copy-on-write semantics for cached ASTs to avoid invalidation blocking reads.

Implementation Guidance

Our Language Engine requires sophisticated parsing and analysis capabilities while maintaining high performance standards. The implementation balances correctness with responsiveness through careful caching and incremental processing strategies.

Technology Recommendations

Component	Simple Option	Advanced Option
Parser	Recursive descent parser with manual AST construction	Tree-sitter incremental parser with grammar file
Symbol Storage	Map-based symbol tables with linear search	B-tree indexed symbol database with bloom filters
Cache Management	Simple LRU cache with memory limits	Multi-tier cache with compression and persistence
Concurrency	Single-threaded with task queue	Actor model with message passing between components

Recommended File Structure

Our Language Engine integrates into the broader LSP server architecture while maintaining clear separation of concerns:

```
project-root/
└── src/
    ├── language-engine/
    │   ├── index.ts           ← LanguageEngine main class
    │   └── parser/
    │       ├── ast-builder.ts    ← AST construction from source text
    │       ├── incremental-parser.ts ← Incremental parsing logic
    │       └── syntax-tree.ts     ← AST node type definitions
    │   └── symbols/
    │       ├── symbol-resolver.ts ← Symbol resolution implementation
    │       ├── scope-manager.ts   ← Scope tree construction and maintenance
    │       └── symbol-index.ts     ← Global symbol indexing across documents
    │   └── cache/
    │       ├── ast-cache.ts      ← Multi-tier AST caching implementation
    │       ├── cache-policy.ts    ← Eviction and promotion policies
    │       └── invalidation.ts    ← Cache invalidation strategies
    │   └── analysis/
    │       ├── semantic-analyzer.ts ← Type checking and error detection
    │       └── diagnostic-collector.ts ← Error and warning aggregation
    └── document-manager/
        └── index.ts           ← Document state management
    └── tests/
        └── language-engine/
            ├── parser.test.ts
            ├── symbols.test.ts
            └── cache.test.ts
```

Infrastructure Starter Code

AST Node Type Definitions (syntax-tree.ts)

```
export interface ASTNode {  
    kind: string;  
    range: Range;  
    parent?: ASTNode;  
    children: ASTNode[];  
}  
  
export interface IdentifierNode extends ASTNode {  
    kind: 'Identifier';  
    name: string;  
}  
  
export interface FunctionDeclarationNode extends ASTNode {  
    kind: 'FunctionDeclaration';  
    name: IdentifierNode;  
    parameters: ParameterNode[];  
    body: BlockStatementNode;  
}  
  
export interface VariableDeclarationNode extends ASTNode {  
    kind: 'VariableDeclaration';  
    declarations: VariableDeclaratorNode[];  
}  
  
export interface VariableDeclaratorNode extends ASTNode {  
    kind: 'VariableDeclarator';  
    id: IdentifierNode;  
    init?: ASTNode;  
}
```

```
export interface BlockStatementNode extends ASTNode {
    kind: 'BlockStatement';
    body: ASTNode[];
}

export interface ParameterNode extends ASTNode {
    kind: 'Parameter';
    name: IdentifierNode;
    type?: ASTNode;
}

// Utility functions for AST traversal

export function walkAST(node: ASTNode, visitor: (node: ASTNode) => boolean | void): void {
    const shouldContinue = visitor(node);
    if (shouldContinue === false) return;

    for (const child of node.children) {
        walkAST(child, visitor);
    }
}

export function findNodeAt(root: ASTNode, position: Position): ASTNode | null {
    let result: ASTNode | null = null;

    walkAST(root, (node) => {
        if (isPositionInRange(position, node.range)) {
            result = node;
            return true; // Continue walking to find more specific nodes
        }
    })
}
```

```
    return false; // Skip this subtree

});

return result;
}

function isPositionInRange(position: Position, range: Range): boolean {
    if (position.line < range.start.line || position.line > range.end.line) {
        return false;
    }

    if (position.line === range.start.line && position.character < range.start.character) {
        return false;
    }

    if (position.line === range.end.line && position.character > range.end.character) {
        return false;
    }

    return true;
}
```

Cache Policy Implementation (cache-policy.ts)

```
interface CacheEntry<T> {  
    value: T;  
    lastAccessed: number;  
    accessCount: number;  
    tier: 'hot' | 'warm' | 'cold';  
}  
  
export class CachePolicy<T> {  
    private entries = new Map<string, CacheEntry<T>>();  
    private hotCapacity = 50;  
    private warmCapacity = 200;  
  
    get(key: string): T | undefined {  
        const entry = this.entries.get(key);  
        if (!entry) return undefined;  
  
        // Update access statistics  
        entry.lastAccessed = Date.now();  
        entry.accessCount++;  
  
        // Consider promotion based on access pattern  
        this.considerPromotion(key, entry);  
  
        return entry.value;  
    }  
  
    set(key: string, value: T, tier: 'hot' | 'warm' | 'cold' = 'hot'): void {  
        // Evict if necessary before adding  
    }  
}
```

```
this.evictIfNecessary(tier);

this.entries.set(key, {
  value,
  lastAccessed: Date.now(),
  accessCount: 1,
  tier
});
}

invalidate(key: string): void {
  this.entries.delete(key);
}

private considerPromotion(key: string, entry: CacheEntry<T>): void {
  const now = Date.now();
  const timeSinceAccess = now - entry.lastAccessed;

  // Promote from cold to warm if accessed frequently
  if (entry.tier === 'cold' && entry.accessCount >= 3) {
    entry.tier = 'warm';
  }

  // Promote from warm to hot if accessed recently and frequently
  if (entry.tier === 'warm' && timeSinceAccess < 300000 && entry.accessCount >= 5) {
    entry.tier = 'hot';
  }
}
```

```
}

private evictIfNecessary(targetTier: 'hot' | 'warm' | 'cold'): void {

    // Count entries by tier

    const tierCounts = { hot: 0, warm: 0, cold: 0 };

    for (const entry of this.entries.values()) {

        tierCounts[entry.tier]++;
    }

    // Evict from target tier if over capacity

    if (targetTier === 'hot' && tierCounts.hot >= this.hotCapacity) {

        this.evictLeastRecentlyUsed('hot');

    } else if (targetTier === 'warm' && tierCounts.warm >= this.warmCapacity) {

        this.evictLeastRecentlyUsed('warm');

    }
}

private evictLeastRecentlyUsed(tier: 'hot' | 'warm' | 'cold'): void {

    let oldestKey = '';
    let oldestTime = Infinity;

    for (const [key, entry] of this.entries) {

        if (entry.tier === tier && entry.lastAccessed < oldestTime) {

            oldestTime = entry.lastAccessed;
            oldestKey = key;
        }
    }
}
```

```
    if (oldestKey) {  
  
        this.entries.delete(oldestKey);  
  
    }  
  
}  
  
}
```

Core Logic Skeleton Code

Main LanguageEngine Class (index.ts)

```
import { TextDocumentItem, Position, Range, Symbol, Diagnostic } from '../types';
```

TYPESCRIPT

```
import { ASTNode } from './parser/syntax-tree';
```

```
import { Scope } from './symbols/scope-manager';
```

```
export class LanguageEngine {
```

```
    private documentASTs = new Map<string, ASTNode>();
```

```
    private documentSymbols = new Map<string, Map<string, Symbol>>();
```

```
    private globalScope: Scope | null = null;
```

```
    constructor() {
```

```
        // TODO 1: Initialize the AST cache with appropriate capacity limits
```

```
        // TODO 2: Set up the global symbol index for cross-document resolution
```

```
        // TODO 3: Configure incremental parsing thresholds and fallback policies
```

```
}
```

```
    async analyzeDocument(document: TextDocumentItem): Promise<void> {
```

```
        // TODO 1: Check if we can use incremental parsing based on document changes
```

```
        // TODO 2: Parse the document source text into an AST using selected strategy
```

```
        // TODO 3: Build symbol table from the AST for this document
```

```
        // TODO 4: Update global symbol index with exported symbols from this document
```

```
        // TODO 5: Store the AST and symbols in appropriate cache tier
```

```
        // TODO 6: Trigger diagnostic analysis for the updated document
```

```
}
```

```
    findSymbolAt(uri: string, position: Position): Symbol | undefined {
```

```
        // TODO 1: Retrieve the AST for the specified document from cache
```

```
        // TODO 2: Find the AST node at the given position using tree traversal
```

```
        // TODO 3: If the node is an identifier, resolve it to its declaration
```

```
// TODO 4: Walk up the scope hierarchy to find the symbol declaration

// TODO 5: Return the Symbol information including range and kind

// Hint: Use findNodeAt() from syntax-tree.ts to locate the node

}

findDefinition(uri: string, position: Position): Range | undefined {

    // TODO 1: Find the symbol at the specified position

    // TODO 2: If symbol is found, return its declaration range

    // TODO 3: Handle cross-document references by searching global symbol index

    // TODO 4: Return undefined if no definition found or symbol is ambiguous

}

getCompletionItems(uri: string, position: Position): CompletionItem[] {

    // TODO 1: Determine the completion context (inside function, class, etc.)

    // TODO 2: Find the scope containing the completion position

    // TODO 3: Gather visible symbols from current scope and parent scopes

    // TODO 4: Include global symbols and imported symbols based on context

    // TODO 5: Filter symbols based on any partially typed prefix

    // TODO 6: Convert symbols to CompletionItem format with labels and documentation

}

getHoverInformation(uri: string, position: Position): string | undefined {

    // TODO 1: Find the symbol at the hover position

    // TODO 2: Retrieve type information and documentation for the symbol

    // TODO 3: Format the information as markdown for display

    // TODO 4: Include signature information for functions and methods

    // TODO 5: Return undefined if no meaningful hover information available
```

```
}

getDiagnostics(uri: string): Diagnostic[] {

    // TODO 1: Retrieve the AST for the document

    // TODO 2: Check for syntax errors in the AST

    // TODO 3: Perform semantic analysis (type checking, undefined variables)

    // TODO 4: Check for style issues and potential bugs

    // TODO 5: Convert analysis results to LSP Diagnostic format

    // TODO 6: Return array of diagnostics sorted by source position

}

private invalidateCache(uri: string): void {

    // TODO 1: Remove AST and symbol table entries for the document

    // TODO 2: Find all documents that import symbols from this document

    // TODO 3: Invalidate symbol resolution caches for dependent documents

    // TODO 4: Update global symbol index to remove exported symbols

    // TODO 5: Trigger re-analysis for documents with broken symbol references

}

}
```

Symbol Resolution Implementation (symbols/symbol-resolver.ts)

```
import { Position, Range, Symbol } from '../../types';
```

TYPESCRIPT

```
import { ASTNode, IdentifierNode } from '../parser/syntax-tree';
```

```
import { Scope } from './scope-manager';
```

```
export class SymbolResolver {
```

```
    private scopes = new Map<string, Scope>(); // document URI -> root scope
```

```
    resolveSymbol(uri: string, identifier: IdentifierNode): Symbol | undefined {
```

```
        // TODO 1: Get the root scope for the document
```

```
        // TODO 2: Find the most specific scope containing the identifier position
```

```
        // TODO 3: Search for the symbol name in the current scope
```

```
        // TODO 4: If not found, recursively search parent scopes
```

```
        // TODO 5: Apply visibility rules (public/private) to validate access
```

```
        // TODO 6: Return the first valid symbol found, or undefined
```

```
}
```

```
    buildSymbolTable(uri: string, ast: ASTNode): Map<string, Symbol> {
```

```
        // TODO 1: Create root scope for the document
```

```
        // TODO 2: Walk the AST to identify scope boundaries (functions, classes, blocks)
```

```
        // TODO 3: For each scope, collect local symbol declarations
```

```
        // TODO 4: Build parent-child relationships between scopes
```

```
        // TODO 5: Create Symbol objects with correct ranges and kinds
```

```
        // TODO 6: Store the scope tree and return flat symbol map
```

```
}
```

```
    private findScopeAt(rootScope: Scope, position: Position): Scope | undefined {
```

```
        // TODO 1: Check if position is within root scope range
```

```
        // TODO 2: Recursively search child scopes for more specific matches
```

```

    // TODO 3: Return the most deeply nested scope containing the position

    // TODO 4: Handle edge cases where position is exactly on scope boundary

}

invalidSymbols(uri: string): void {

    // TODO 1: Remove scope tree for the document

    // TODO 2: Clear any cached symbol resolution results

    // TODO 3: Notify dependent components of symbol changes

}

}

```

Language-Specific Hints

TypeScript-Specific Implementation Notes:

- Use the TypeScript compiler API (`ts.createSourceFile`) for robust parsing instead of building a custom parser—it handles all edge cases and provides excellent error recovery
- Leverage `ts.forEachChild()` for efficient AST traversal without manual recursion
- The TypeScript `SymbolTable` and `TypeChecker` APIs provide sophisticated symbol resolution that handles complex cases like union types and generics
- Use `ts.getPositionOfLineAndCharacter()` and `ts.getLineAndCharacterOfPosition()` for accurate position conversion between LSP and TypeScript coordinate systems

Performance Optimization Tips:

- Implement `WeakMap` for AST node caching to allow garbage collection of unused nodes
- Use `Set` instead of `Array` for symbol visibility checks to achieve O(1) lookup performance
- Consider implementing AST node interning to reduce memory usage for common node patterns
- Use `Object.freeze()` on immutable AST nodes to enable safe sharing between cache tiers

Milestone Checkpoints

After Implementing Basic Parsing (Milestone 3 foundation):

Run the test command: `npm test -- --grep "Language Engine"` and verify:

- Parsing a simple function declaration creates correct AST structure with function, parameter, and body nodes
- Symbol table contains entries for function name and parameter names with correct source ranges
- Cache stores the AST and retrieves it on subsequent requests without re-parsing

After Implementing Symbol Resolution (Milestone 3 core):

Test manually by creating a sample document with nested scopes:

```
function outer(param: string) {  
    const local = "value";  
  
    function inner() {  
        return param + local; // Should resolve both identifiers  
    }  
}
```

TYPESCRIPT

Verify that:

- `findSymbolAt()` correctly identifies `param` as a parameter symbol when cursor is on the reference
- `findDefinition()` returns the parameter declaration range for `param` reference
- Symbol resolution respects scope boundaries and doesn't find symbols outside their visibility

After Implementing Completion (Milestone 3 completion):

Position cursor inside the `inner` function and verify completion includes:

- `param` from outer function scope
- `local` from outer function scope
- Global symbols and built-in functions
- No symbols from sibling scopes or unrelated functions

Feature Provider Design

Milestone(s): This section corresponds primarily to Milestone 3 (Language Features) and Milestone 4 (Diagnostics & Code Actions), implementing the core IDE features that developers experience when using a language server.

Think of feature providers as specialized consultants in a law firm. The Language Engine is like the firm's massive legal research department that knows all the laws and precedents (symbols, types, scopes). Each feature provider is a specialist consultant who takes client questions ("What are my options here?" for completion, "What does this mean?" for hover, "Where is this defined?" for go-to-definition) and uses the research department's knowledge to provide focused, actionable answers. Just as each consultant has their own expertise and methodology for answering questions, each feature provider implements a specific IDE capability using the Language Engine's analysis capabilities.

The Feature Provider layer sits at the top of our LSP server architecture, directly interfacing with client requests and translating them into concrete IDE functionality. These providers coordinate with the Language Engine to perform symbol resolution, scope analysis, and semantic understanding, then format the results according to LSP protocol specifications. Each provider specializes in a particular aspect of the developer experience, from intelligent code completion to real-time error reporting.

This design separates the concerns of language analysis (handled by the Language Engine) from user interface concerns (handled by the Feature Providers). The Language Engine focuses purely on understanding the code's semantic structure, while each Feature Provider focuses on presenting that understanding in a way that's useful for a specific IDE feature. This separation allows us to optimize each provider independently and makes it easier to add new features without modifying the core language analysis logic.

Symbol Resolution Process

Start

Find Identifier
at Position

Walk Up AST
to Find Enclosing
Scopes

Has Parent
Scope?

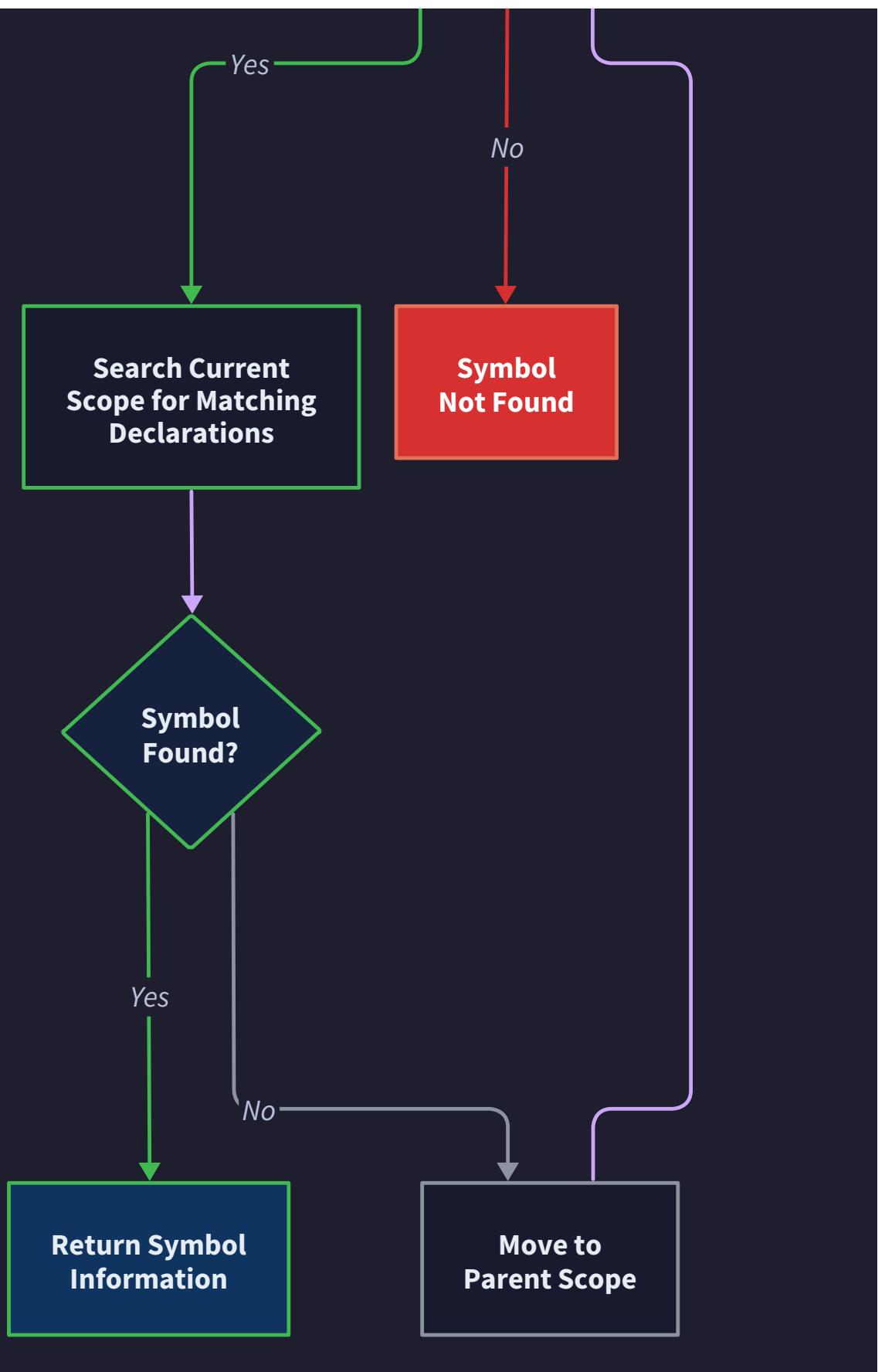
Symbol Resolution Process

Start

Find Identifier
at Position

Walk Up AST
to Find Enclosing
Scopes

Has Parent
Scope?



Completion Provider

The Completion Provider transforms the "What could I write here?" question into a ranked list of contextually appropriate suggestions. Think of it like an experienced pair-programming partner who, when you pause mid-sentence, can suggest the most likely ways to complete your thought based on what's already available in scope, what you've imported, and what makes sense at that particular position in the code.

The provider operates on the principle of **contextual filtering** - starting with everything that could theoretically be valid at a position, then progressively narrowing down to what's actually useful. This happens through several layers of analysis: syntactic context (are we in an expression, a statement, a type annotation?), semantic context (what types are expected here?), and lexical context (what symbols are visible in the current scope?).

Decision: Lazy vs Eager Completion Generation

- **Context:** Completion requests are frequent and must respond within 100-200ms to feel responsive. We need to decide whether to pre-compute all possible completions for a document or compute them on-demand per request.
- **Options Considered:**
 - **Eager:** Pre-compute all completions when document changes, cache by position
 - **Lazy:** Compute completions on-demand for each request
 - **Hybrid:** Pre-compute symbol tables, compute completions on-demand
- **Decision:** Hybrid approach with lazy completion generation
- **Rationale:** Pre-computing completions for every possible position is memory-intensive and most positions never need completion. However, symbol table analysis is expensive enough to warrant caching. The hybrid approach gives us fast symbol lookup with minimal memory overhead.
- **Consequences:** We cache ASTs and symbol tables but generate completion lists on-demand, allowing us to incorporate real-time context like partial identifiers and cursor position.

Completion Strategy	Memory Usage	Response Time	Accuracy	Chosen?
Eager pre-computation	High ($O(\text{positions} \times \text{symbols})$)	Fastest (cache lookup)	Medium (stale context)	✗
Pure lazy generation	Low ($O(\text{current scope})$)	Slowest (full analysis)	Highest (real-time context)	✗
Hybrid (cached symbols, lazy completion)	Medium ($O(\text{symbols})$)	Fast (symbol lookup + filtering)	High (current context)	✓

The completion generation algorithm follows these steps:

1. **Position Analysis:** Determine the syntactic context of the cursor position by walking up the AST to find the enclosing node. This tells us whether we're completing an identifier, a member access, a function call, or a type annotation.

2. **Partial Identifier Extraction:** If the user has already typed part of an identifier, extract that prefix to use for filtering. Handle cases where the cursor is in the middle of an existing identifier.
3. **Scope Resolution:** Starting from the current position, walk up through enclosing scopes to collect all visible symbols. This includes local variables, function parameters, imported modules, and global declarations.
4. **Type-Based Filtering:** If we can determine the expected type at this position (e.g., we're on the right side of an assignment), filter completions to only include symbols of compatible types.
5. **Prefix Matching:** Filter the collected symbols to only those that match the partial identifier prefix, using fuzzy matching algorithms that handle typos and abbreviations.
6. **Ranking and Sorting:** Sort completions by relevance using factors like scope proximity (local variables ranked higher than globals), usage frequency, and alphabetical order as tiebreakers.
7. **Detail Generation:** For each completion item, generate appropriate documentation, type information, and additional text edits needed for insertion.

The `CompletionProvider` maintains several key responsibilities that require careful coordination:

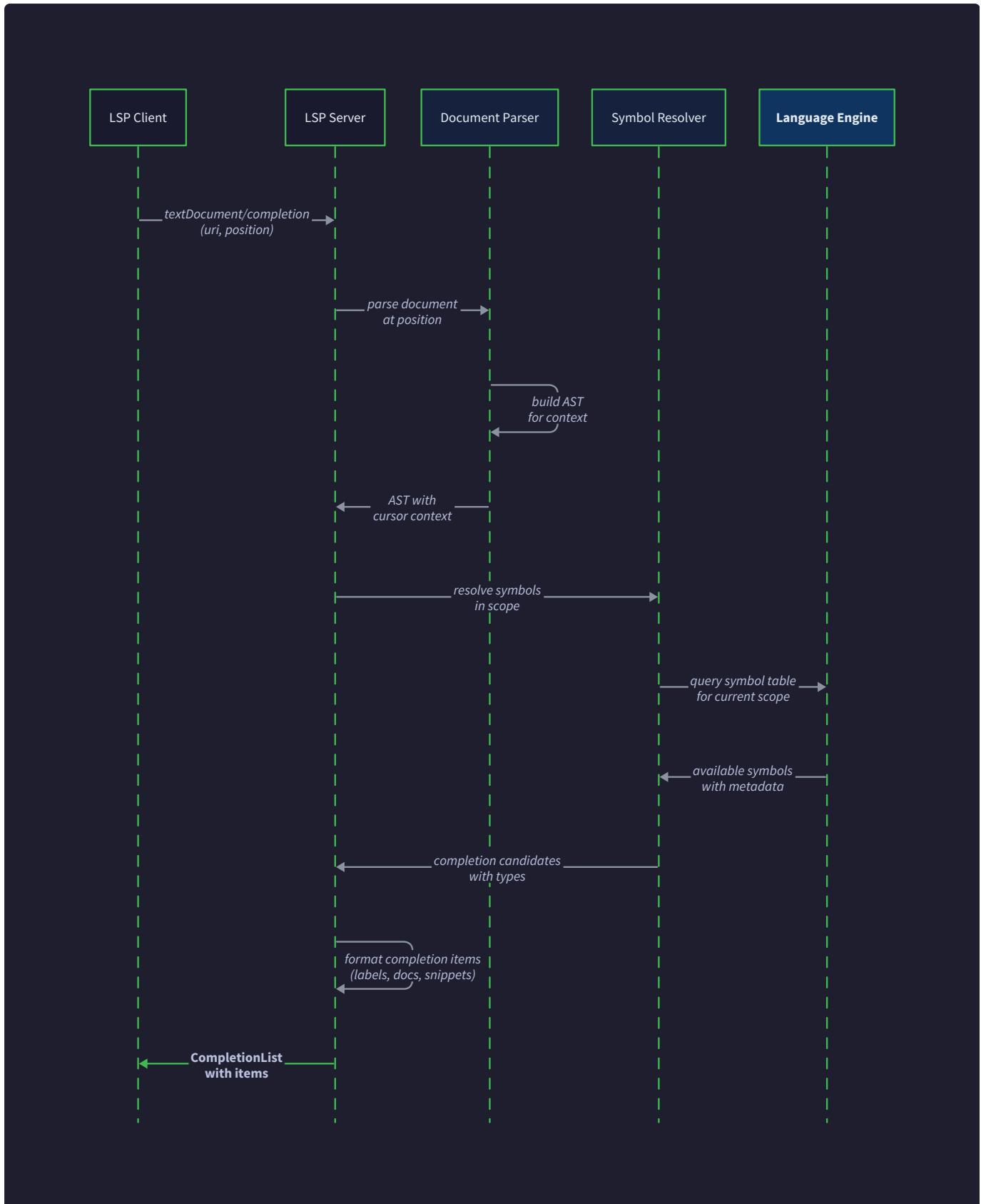
Method Name	Parameters	Returns	Description
<code>provideCompletions</code>	<code>uri: string,</code> <code>position:</code> <code>Position, context:</code> <code>CompletionContext</code>	<code>Promise<CompletionItem[]></code>	Main entry point that coordinates the completion generation pipeline
<code>getSymbolsInScope</code>	<code>uri: string,</code> <code>position:</code> <code>Position</code>	<code>Symbol[]</code>	Collects all symbols visible at the given position by walking scope hierarchy
<code>filterByPrefix</code>	<code>symbols:</code> <code>Symbol[], prefix:</code> <code>string</code>	<code>Symbol[]</code>	Applies fuzzy matching to filter symbols by partial identifier
<code>rankCompletions</code>	<code>items:</code> <code>CompletionItem[],</code> <code>context:</code> <code>CompletionContext</code>	<code>CompletionItem[]</code>	Sorts completions by relevance score
<code>generateCompletionItem</code>	<code>symbol: Symbol,</code> <code>prefix: string</code>	<code>CompletionItem</code>	Converts a symbol into a formatted completion item with documentation

The completion context includes several pieces of information that influence the generation process:

Context Field	Type	Description
<code>triggerKind</code>	<code>number</code>	Whether completion was triggered by typing, explicit invocation, or trigger character
<code>triggerCharacter</code>	<code>string?</code>	The character that triggered completion (e.g., '.' for member access)
<code>partialIdentifier</code>	<code>string</code>	Any text the user has already typed at this position
<code>expectedType</code>	<code>string?</code>	The expected type at this position if determinable from context
<code>enclosingNode</code>	<code>ASTNode</code>	The AST node that contains the completion position

Member access completion (triggered by typing '.') requires special handling because it involves resolving the type of the expression to the left of the dot, then providing completions based on that type's members. This is more complex than simple identifier completion because it requires type inference capabilities from the Language Engine.

The key insight for responsive completion is that users typically want completions for recently-used symbols and symbols in the immediate scope. We optimize for this by checking local scope first and can even return partial results immediately while continuing to search broader scopes in the background.



Hover Provider

The Hover Provider acts like a knowledgeable code reviewer who can instantly explain what any piece of code means when you point at it. Think of it as having access to all the documentation, type information, and

contextual details about every symbol in your codebase, delivered in a tooltip format that doesn't interrupt your workflow.

Unlike completion which deals with future possibilities, hover deals with present reality - it explains what's already there. The user points to a symbol, and the provider must quickly determine what that symbol represents, where it was declared, what type it has, and any available documentation, then present this information in a formatted, readable way.

The hover response format uses **MarkupContent** to provide rich formatting including syntax highlighting, markdown formatting, and structured information display. This allows the provider to present complex type signatures, documentation strings, and usage examples in a way that's both human-readable and visually appealing.

Decision: Hover Content Structure

- **Context:** Hover responses can include various types of information (type signature, documentation, examples, source location). We need to decide how to structure and prioritize this information.
- **Options Considered:**
 - **Minimal:** Just type signature or brief description
 - **Comprehensive:** Type signature + documentation + examples + source location
 - **Adaptive:** Content varies based on symbol type and available information
- **Decision:** Adaptive comprehensive approach
- **Rationale:** Different symbols benefit from different information. A function needs signature + docs, a variable needs type + current value, a type needs definition + usage. However, we always want to be comprehensive when information is available.
- **Consequences:** Hover responses are more useful but require more complex generation logic and may be slower for symbols with extensive documentation.

The hover generation process follows a systematic approach to information gathering and presentation:

1. **Symbol Identification:** Use the Language Engine to identify the symbol at the hover position. This requires precise position mapping from LSP coordinates to AST nodes, handling edge cases like whitespace, comments, and multi-character operators.
2. **Declaration Resolution:** Once we have the symbol, resolve it to its declaration site. This may involve following import chains, resolving through alias declarations, or finding the original definition in cases of symbol shadowing.
3. **Type Information Extraction:** Determine the symbol's type information. For variables this is their declared or inferred type, for functions it's their signature, for types it's their definition, for modules it's their exported interface.
4. **Documentation Retrieval:** Look for associated documentation including doc comments, inline comments, and any external documentation references. Parse and format this documentation for display.

5. **Context Information:** Gather additional contextual information like parameter names for function calls, possible values for enums, or usage examples for complex types.

6. **Markup Generation:** Format all gathered information into MarkupContent with appropriate syntax highlighting, section headers, and structured layout.

The `HoverProvider` interface defines the core operations needed for hover functionality:

Method Name	Parameters	Returns	Description
<code>provideHover</code>	<code>uri: string,</code> <code>position: Position</code>	<code>Promise<Hover?></code>	Main entry point that generates hover information for the position
<code>resolveSymbolDeclaration</code>	<code>uri: string,</code> <code>symbol: Symbol</code>	<code>Promise<DeclarationInfo></code>	Finds the original declaration site for a symbol
<code>extractTypeInformation</code>	<code>symbol: Symbol</code>	<code>TypeInfo</code>	Generates formatted type signature and related type information
<code>gatherDocumentation</code>	<code>symbol: Symbol</code>	<code>DocumentationInfo</code>	Collects and formats all available documentation for the symbol
<code>formatHoverContent</code>	<code>typeInfo: TypeInfo,</code> <code>docs: DocumentationInfo</code>	<code>MarkupContent</code>	Combines all information into formatted hover response

The hover response structure varies based on the type of symbol being hovered over:

Symbol Type	Content Structure	Example Information
Variable	<code>type: Type\nvalue: Value\nDeclared at: Location</code>	<code>count: number\nvalue: 42\nDeclared at line 15</code>
Function	<code>function signature\nparameters\ndocumentation\nexamples</code>	<code>calculateTotal(items: Item[]): number\nCalculates total price including tax</code>
Type	<code>type definition\nproperties/methods\nusage examples</code>	<code>interface User { name: string; email: string; }</code>
Import	<code>imported from: module\ntype: imported type\navailable exports</code>	<code>imported from './utils'\n<code>type: function\nexports: calculateTotal, formatCurrency</code></code>

Special considerations for hover include handling **overloaded functions** where multiple signatures exist, **generic types** where type parameters need to be displayed clearly, and **partial type information** where the Language Engine cannot fully infer types but can provide useful partial information.

A critical performance consideration is that hover requests are extremely frequent during active coding sessions. Users often hover multiple times per second while exploring unfamiliar code. The hover provider must respond within 50-100ms to feel instantaneous, which means aggressive caching of symbol resolution and type information.

Definition Provider

The Definition Provider implements the "go-to-definition" feature that developers rely on for code navigation. Think of it as a librarian who can instantly tell you exactly which shelf, which book, and which page contains the original definition of any concept you encounter while reading. When a developer clicks on a function call, variable reference, or type name, this provider must quickly locate the exact source location where that symbol was originally declared.

The challenge lies in handling the complexity of modern programming languages where a single identifier might refer to different symbols depending on context, scope, and even generic type instantiation. The provider must implement sophisticated **symbol resolution** that accounts for scope hierarchy, import/export relationships, and language-specific binding rules.

Decision: Multi-Definition Handling

- **Context:** Many symbols can have multiple definitions (overloaded functions, partial classes, module exports). We need to decide whether to return the first definition, all definitions, or the "best" definition.
- **Options Considered:**
 - **Single Definition:** Return only the most relevant definition
 - **Multiple Definitions:** Return all possible definitions and let the client choose
 - **Hierarchical:** Return primary definition immediately, provide secondary definitions on demand
- **Decision:** Multiple definitions with primary ranking
- **Rationale:** Modern IDEs handle multiple definitions well (showing a list or peek window). Developers often want to see all definitions, especially for overloaded functions or interface implementations.
- **Consequences:** Slightly more complex implementation but much more useful for developers working with polymorphic code or large codebases with symbol reuse.

The definition resolution process involves several layers of analysis that must work together seamlessly:

1. **Precise Symbol Location:** Convert the LSP position into an AST node, ensuring we identify the exact symbol token rather than surrounding syntax. This requires careful handling of compound identifiers, member access chains, and qualified names.
2. **Reference vs Declaration Distinction:** Determine whether the position points to a symbol declaration or a reference to that symbol. If it's already a declaration, we might want to find all references instead, or find the interface that this declaration implements.
3. **Scope Chain Resolution:** Walk up the scope hierarchy to find all possible bindings for this identifier name. Handle cases where the same name might be bound in multiple scopes (shadowing) or where the binding depends on execution context.
4. **Cross-File Resolution:** For symbols defined in other files, resolve import/export relationships to locate the actual definition file and position. This requires maintaining an accurate module dependency graph and handling different import styles (relative paths, absolute imports, aliased imports).
5. **Generic and Template Handling:** For generic types and template instantiations, navigate to the base generic definition rather than getting lost in instantiated versions. Provide enough context to understand which generic parameters are being used.
6. **Ranking and Filtering:** When multiple definitions exist, rank them by relevance - typically prioritizing the most direct definition, then interface definitions, then implementations.

The `DefinitionProvider` coordinates these complex resolution steps through a clean interface:

Method Name	Parameters	Returns	Description
<code>provideDefinition</code>	<code>uri: string,</code> <code>position:</code> <code>Position</code>	<code>Promise<Location[]></code>	Main entry point returning all definition locations for the symbol
<code>resolveSymbolReference</code>	<code>uri: string,</code> <code>position:</code> <code>Position</code>	<code>Promise<SymbolReference></code>	Identifies the symbol and its binding context at the position
<code>findDeclarationSites</code>	<code>symbolRef:</code> <code>SymbolReference</code>	<code>Promise<DeclarationSite[]></code>	Locates all places where this symbol is declared
<code>resolveImportedSymbol</code>	<code>importInfo:</code> <code>ImportInfo,</code> <code>symbolName:</code> <code>string</code>	<code>Promise<Location></code>	Follows import chains to find the ultimate definition
<code>rankDefinitions</code>	<code>definitions:</code> <code>DeclarationSite[]</code>	<code>DeclarationSite[]</code>	Orders multiple definitions by relevance and directness

The resolution process must handle several complex scenarios that frequently occur in real codebases:

Scenario	Challenge	Resolution Strategy
Overloaded Functions	Multiple valid definitions with different signatures	Return all definitions, ranked by signature match if call site provides context
Interface Implementations	Symbol could refer to interface declaration or concrete implementation	Return interface first, then implementations, clearly labeled
Re-exported Symbols	Symbol imported and then exported from intermediate modules	Follow the full chain to the original definition, not intermediate re-exports
Generic Instantiations	Reference to <code>List<string></code> should go to <code>List<T></code> definition	Resolve to the generic type definition with type parameter information
Aliased Imports	<code>import { foo as bar }</code> creates name mapping	Resolve through the alias mapping to find the original symbol
Scoped Symbols	Same name bound in multiple nested scopes	Find the binding that's actually visible at the reference site

Cross-file resolution presents particular challenges because it requires maintaining accurate dependency information and handling cases where files might not be currently open in the editor. The provider must coordinate with the Document Manager to ensure that imported files are parsed and analyzed as needed, potentially triggering background analysis of unopened files.

The key insight for definition resolution is that most requests are for recently-accessed symbols within the same file or closely-related files. We optimize for this by maintaining hot caches for intra-file symbol tables and recently-resolved cross-file references, while falling back to slower but comprehensive analysis for rarely-accessed symbols.

Performance considerations are crucial because developers expect go-to-definition to be nearly instantaneous. The provider implements several optimization strategies including symbol table caching, import resolution memoization, and lazy loading of definition details. For large codebases, we may need to implement background indexing to pre-resolve common cross-file references.

Diagnostic Provider

The Diagnostic Provider serves as the LSP server's quality assurance system, continuously monitoring code for errors, warnings, and potential improvements. Think of it as a combination of a vigilant proofreader who catches typos and grammatical errors, a code reviewer who spots logical problems, and a style guide enforcer who ensures consistency. Unlike other providers that respond to explicit user requests, the diagnostic provider runs proactively, publishing findings to the client whenever document content changes.

The provider operates on multiple analysis levels, from basic syntax validation to sophisticated semantic analysis and even style checking. Each level provides different types of insights and operates with different performance

characteristics. The challenge is to balance comprehensiveness with responsiveness - developers expect to see error indicators within seconds of typing, but comprehensive analysis can be computationally expensive.

Decision: Incremental vs Full Document Analysis

- **Context:** Diagnostics must be published quickly after document changes, but full document analysis can be slow for large files. We need to balance responsiveness with accuracy.
- **Options Considered:**
 - **Full Analysis:** Re-analyze entire document on every change
 - **Incremental Analysis:** Analyze only changed regions and their dependencies
 - **Tiered Analysis:** Fast syntax check immediately, slower semantic analysis in background
- **Decision:** Tiered analysis with incremental optimization
- **Rationale:** Users need immediate feedback for syntax errors (typos, missing brackets) but can tolerate slight delays for complex semantic warnings. Incremental optimization provides the best of both worlds when feasible.
- **Consequences:** More complex implementation but much better user experience. Fast feedback loop for common errors, comprehensive analysis without blocking the editor.

Analysis Strategy	Speed	Accuracy	Resource Usage	User Experience	Chosen?
Full re-analysis	Slow	Highest	High CPU	Laggy, but complete results	✗
Incremental only	Fast	Medium	Low CPU	Responsive, but may miss dependencies	✗
Tiered (syntax + semantic)	Fast initial, slower complete	High	Medium CPU	Best of both: immediate + comprehensive	✓

The diagnostic generation pipeline processes documents through multiple analysis phases:

1. **Syntax Analysis Phase:** Performed immediately when document content changes. Catches parsing errors, missing brackets, invalid tokens, and other structural problems that prevent the document from forming a valid AST. These diagnostics have `Error` severity because they prevent further analysis.
2. **Semantic Analysis Phase:** Performed after syntax analysis succeeds. Validates type compatibility, checks variable declarations and usage, verifies function calls match signatures, and ensures imported symbols exist. These diagnostics typically have `Error` or `Warning` severity based on whether they prevent code execution.
3. **Style and Convention Phase:** Performed in the background after semantic analysis. Checks coding style, naming conventions, unused variables, and other code quality issues. These diagnostics usually have `Warning` or `Information` severity since they don't affect functionality.

4. Cross-File Analysis Phase: Performed periodically or when dependencies change. Validates that imported symbols still exist, checks for breaking changes in dependencies, and identifies inconsistencies across related files. These diagnostics help maintain codebase-wide consistency.

The `DiagnosticProvider` coordinates these analysis phases and manages the publishing of results:

Method Name	Parameters	Returns	Description
<code>publishDiagnostics</code>	<code>uri: string,</code> <code>document: TextDocumentItem</code>	<code>Promise<void></code>	Main entry point that triggers analysis and publishes results to client
<code>analyzeSyntax</code>	<code>uri: string, text: string</code>	<code>Diagnostic[]</code>	Fast syntax validation returning structural errors
<code>analyzeSemantics</code>	<code>uri: string, ast: ASTNode</code>	<code>Promise<Diagnostic[]></code>	Semantic validation using symbol resolution and type checking
<code>analyzeStyle</code>	<code>uri: string, ast: ASTNode</code>	<code>Promise<Diagnostic[]></code>	Style and convention checking for code quality
<code>analyzeCrossFile</code>	<code>uri: string</code>	<code>Promise<Diagnostic[]></code>	Cross-file consistency and dependency validation
<code>mergeDiagnostics</code>	<code>diagnosticSets: Diagnostic[][]</code>	<code>Diagnostic[]</code>	Combines results from multiple analysis phases, handling overlaps

Diagnostic severity levels follow LSP standards but require careful consideration of what constitutes each level:

Severity Level	Numeric Value	Use Cases	Examples
Error	1	Prevents compilation/execution	Syntax errors, undefined variables, type mismatches
Warning	2	Potential problems, deprecated usage	Unused variables, deprecated API calls, potential null references
Information	3	Suggestions, informational notices	Code style suggestions, performance hints
Hint	4	Very minor suggestions	Extra whitespace, redundant code

The diagnostic data structure includes rich information beyond just the error message:

Diagnostic Field	Type	Description
range	Range	Source location where the diagnostic applies
severity	number	Error level (1=Error, 2=Warning, 3=Information, 4=Hint)
message	string	Human-readable description of the issue
source	string	Name of the tool/analyzer that generated this diagnostic
code	string number	Machine-readable diagnostic code for programmatic handling
codeDescription	object	Link to documentation explaining the diagnostic
tags	number[]	Additional metadata (deprecated, unused, etc.)
relatedInformation	DiagnosticRelatedInformation[]	Additional locations relevant to this diagnostic

Error recovery and partial analysis capabilities are crucial for maintaining useful diagnostics even when the code is in a temporarily invalid state (which is common during active editing). The provider must be robust enough to provide meaningful feedback even when the AST is malformed or incomplete.

A critical insight is that developers care more about the first few errors than about comprehensive error reporting. If there are 50 type errors in a file, showing the first 5-10 in dependency order is more useful than overwhelming the user with the complete list. We prioritize early errors and errors that are likely to cause cascading failures.

Performance optimization focuses on avoiding redundant work across multiple analysis phases. The provider maintains caches for syntax trees, symbol tables, and previous diagnostic results, invalidating only the portions affected by document changes. For large files, we may implement streaming analysis that publishes partial results as they become available.

Common Pitfalls

⚠ Pitfall: Stale Symbol Resolution

A frequent mistake is caching symbol resolution results without proper invalidation when documents change. For example, if function `calculateTotal` is renamed to `computeTotal` in another file, completion and hover providers may continue suggesting the old name until the cache is cleared. This happens because the providers cache resolved symbols for performance but forget to invalidate caches when dependencies change.

Why it's wrong: Users see outdated suggestions that reference symbols that no longer exist, leading to confusing error messages when they accept the suggestions.

How to fix: Implement dependency tracking between files and invalidate caches transitively. When document A changes, invalidate caches for all documents that import from A. Use document version numbers to detect when cached data is stale.

⚠ Pitfall: Position Coordinate Confusion

LSP uses UTF-16 code unit offsets for character positions, but many language parsers use byte offsets or UTF-8 character offsets. Mixing these coordinate systems leads to off-by-one errors or completely wrong symbol resolution, especially with Unicode characters.

Why it's wrong: Hover requests point to the wrong symbols, completion triggers at incorrect positions, and go-to-definition jumps to nearby but incorrect locations. This is particularly problematic with emoji, accented characters, or non-Latin scripts.

How to fix: Always convert between coordinate systems at the boundary between LSP messages and internal analysis. Maintain consistent coordinate systems within each component, and use explicit conversion functions (`positionToOffset`, `offsetToPosition`) that handle UTF-16 encoding correctly.

⚠ Pitfall: Blocking the Main Thread

Feature providers often perform expensive operations like AST traversal, symbol resolution, or type inference. If these operations run synchronously on the main event loop, they block other LSP requests, making the editor feel unresponsive.

Why it's wrong: Users experience typing delays, completion popups that take seconds to appear, and general editor sluggishness. This is especially problematic during document changes when multiple providers may be running simultaneously.

How to fix: Use asynchronous processing with proper concurrency control. Implement background analysis with result caching, prioritize user-facing requests over background analysis, and consider request debouncing for rapidly-changing documents.

⚠ Pitfall: Over-Aggressive Completion Filtering

Attempting to be too smart about completion filtering can result in hiding useful completions. For example, filtering out all non-matching types when the expected type isn't certain, or being too strict about scope visibility rules in dynamic languages.

Why it's wrong: Users don't see completions they expect, leading them to think the LSP server doesn't understand their code structure or that symbols aren't available when they actually are.

How to fix: Err on the side of showing more completions rather than fewer. Use ranking instead of filtering - show less relevant items at the bottom of the list rather than hiding them completely. Provide configuration options for completion aggressiveness.

Implementation Guidance

The Feature Provider implementation requires careful coordination between multiple analysis systems and caching layers. Each provider specializes in a specific type of user interaction but shares common infrastructure for symbol resolution and document analysis.

Technology Recommendations:

Component	Simple Option	Advanced Option
Completion Engine	Linear search through symbol table + prefix matching	Trie-based symbol indexing with fuzzy matching algorithm
Type Information	Simple string-based type names	Rich type objects with generic parameter tracking
Documentation	Parse comment blocks with regex	Structured documentation parser with markdown support
Symbol Caching	In-memory Map with manual invalidation	LRU cache with dependency-based invalidation
Background Analysis	setTimeout-based delays	Worker threads or process pool for CPU-intensive analysis

Recommended File Structure:

```
src/providers/
  completion/
    completion-provider.ts      ← Main completion logic
    completion-context.ts       ← Context analysis and filtering
    completion-ranking.ts       ← Relevance scoring and sorting
    completion-items.ts         ← CompletionItem generation and formatting
  hover/
    hover-provider.ts          ← Main hover logic
    hover-formatter.ts          ← MarkupContent generation
    type-information.ts         ← Type signature extraction
  definition/
    definition-provider.ts      ← Main definition logic
    symbol-resolver.ts          ← Cross-file symbol resolution
    import-resolver.ts          ← Import/export chain following
  diagnostics/
    diagnostic-provider.ts      ← Main diagnostic coordination
    syntax-analyzer.ts          ← Fast syntax validation
    semantic-analyzer.ts        ← Type checking and semantic validation
    style-analyzer.ts           ← Code style and convention checking
  shared/
    symbol-cache.ts             ← Shared symbol resolution caching
    position-utils.ts            ← LSP position conversion utilities
    formatting-utils.ts          ← Shared text formatting helpers
```

Infrastructure Starter Code (Complete):

```
export interface Position {  
    line: number;  
    character: number;  
}  
  
export interface Range {  
    start: Position;  
    end: Position;  
}  
  
/**  
 * Converts LSP position (line/character) to byte offset in document text.  
 * LSP uses UTF-16 code units for character positions.  
 */  
  
export function positionToOffset(text: string, position: Position): number {  
    const lines = text.split('\n');  
    let offset = 0;  
  
    // Add lengths of all lines before target line  
    for (let i = 0; i < position.line && i < lines.length; i++) {  
        offset += Buffer.from(lines[i] + '\n', 'utf16le').length / 2;  
    }  
  
    // Add character offset within target line  
    if (position.line < lines.length) {  
        const lineText = lines[position.line];  
        const linePrefix = lineText.substring(0, position.character);  
    }  
}
```

```
    offset += Buffer.from(linePrefix, 'utf16le').length / 2;

}

return offset;
}

/**

 * Converts byte offset to LSP position (line/character).

 */

export function offsetToPosition(text: string, offset: number): Position {

const lines = text.split('\n');

let currentOffset = 0;

for (let lineIndex = 0; lineIndex < lines.length; lineIndex++) {

const line = lines[lineIndex];

const lineLength = Buffer.from(line + '\n', 'utf16le').length / 2;

if (currentOffset + lineLength > offset) {

// Target offset is within this line

const characterOffset = offset - currentOffset;

const linePrefix = Buffer.from(line, 'utf16le').toString('utf16le', 0, characterOffset * 2);

return {

line: lineIndex,

character: linePrefix.length

};

}

}
```

```
        currentOffset += lineLength;

    }

    // offset beyond end of document

    return {
        line: Math.max(0, lines.length - 1),
        character: lines[lines.length - 1]?.length || 0
    };
}

/**

 * Checks if position is within range.

 */

export function positionInRange(position: Position, range: Range): boolean {

    if (position.line < range.start.line || position.line > range.end.line) {

        return false;
    }

    if (position.line === range.start.line && position.character < range.start.character) {

        return false;
    }

    if (position.line === range.end.line && position.character > range.end.character) {

        return false;
    }

    return true;
}
```

}

```
// src/providers/shared/symbol-cache.ts
```

TYPESCRIPT

```
export interface CacheEntry<T> {  
  
    value: T;  
  
    lastAccessed: number;  
  
    accessCount: number;  
  
    tier: string;  
  
}  
  
export class SymbolCache<T> {  
  
    private cache = new Map<string, CacheEntry<T>>();  
  
    private readonly hotCapacity = 100;  
  
    private readonly warmCapacity = 500;  
  
  
    set(key: string, value: T): void {  
  
        const entry: CacheEntry<T> = {  
  
            value,  
  
            lastAccessed: Date.now(),  
  
            accessCount: 1,  
  
            tier: 'hot'  
  
        };  
  
  
        this.cache.set(key, entry);  
  
        this.evictIfNeeded();  
  
    }  
  
  
    get(key: string): T | undefined {  
  
        const entry = this.cache.get(key);  
  
        if (!entry) return undefined;
```

```
entry.lastAccessed = Date.now();

entry.accessCount++;

return entry.value;

}

invalidate(keyPattern: string): void {

const regex = new RegExp(keyPattern);

for (const [key] of this.cache) {

if (regex.test(key)) {

this.cache.delete(key);

}

}

}

private evictIfNeeded(): void {

if (this.cache.size <= this.warmCapacity) return;

const entries = Array.from(this.cache.entries());

entries.sort(([a], [b]) => a.lastAccessed - b.lastAccessed);

const toRemove = entries.slice(0, entries.length - this.hotCapacity);

for (const [key] of toRemove) {

this.cache.delete(key);

}

}
```

}

Core Logic Skeletons (TODOs only):

```
// src/providers/completion/completion-provider.ts
```

TYPESCRIPT

```
export interface CompletionItem {  
  
    label: string;  
  
    kind?: number;  
  
    detail?: string;  
  
    documentation?: string;  
  
    insertText?: string;  
  
}
```

```
export interface CompletionContext {  
  
    triggerKind: number;  
  
    triggerCharacter?: string;  
  
    partialIdentifier: string;  
  
    enclosingNode: ASTNode;  
  
}
```

```
export class CompletionProvider {  
  
    constructor(  
  
        private languageEngine: LanguageEngine,  
  
        private symbolCache: SymbolCache<Symbol[]>  
    ) {}
```

```
/**
```

```
* Provides context-aware completions for the given position.
```

```
* Returns completions ranked by relevance.
```

```
*/
```

```
async provideCompletions(uri: string, position: Position): Promise<CompletionItem[]> {  
  
    // TODO 1: Use languageEngine.findNodeAt to get AST node at position
```

```

// TODO 2: Determine completion context (identifier, member access, etc.)

// TODO 3: Extract partial identifier if user has started typing

// TODO 4: Get symbols in scope using getSymbolsInScope

// TODO 5: Filter symbols by prefix using filterByPrefix

// TODO 6: Generate CompletionItem objects using generateCompletionItem

// TODO 7: Rank and sort completions using rankCompletions

// TODO 8: Return top 50 completions to avoid overwhelming the user

// Hint: Check triggerCharacter for member access completion (.)

// Hint: Use symbol.kind to set appropriate CompletionItemKind

}

/***
 * Collects all symbols visible at the given position by walking scope hierarchy.
 */
private async getSymbolsInScope(uri: string, position: Position): Promise<Symbol[]> {

    // TODO 1: Check symbolCache for cached symbols at this position

    // TODO 2: Use languageEngine.findSymbolAt to get current scope

    // TODO 3: Walk up scope hierarchy collecting symbols from each level

    // TODO 4: Include imported symbols from module imports

    // TODO 5: Include global/builtin symbols available in this language

    // TODO 6: Cache result with appropriate invalidation key

    // TODO 7: Return combined symbol list with scope proximity info

    // Hint: Prioritize local scope symbols over global ones

    // Hint: Handle symbol shadowing - inner scope symbols hide outer ones

}

/***

```

```
* Filters symbols by partial identifier using fuzzy matching.

*/
private filterByPrefix(symbols: Symbol[], prefix: string): Symbol[] {

    // TODO 1: If prefix is empty, return all symbols

    // TODO 2: Apply exact prefix matching first (highest priority)

    // TODO 3: Apply case-insensitive prefix matching

    // TODO 4: Apply fuzzy matching for abbreviations (e.g., "calc" matches "calculateTotal")

    // TODO 5: Apply substring matching as fallback

    // TODO 6: Return filtered list maintaining original symbol order within each match
category

    // Hint: Use symbol.name for matching, not symbol.label

    // Hint: Consider acronym matching (e.g., "CT" matches "calculateTotal")
}

}
```

```
export interface Hover {  
  contents: MarkupContent;  
  range?: Range;  
}  
  
export interface MarkupContent {  
  kind: 'plaintext' | 'markdown';  
  value: string;  
}  
  
export class HoverProvider {  
  constructor(private languageEngine: LanguageEngine) {}  
  
  /**  
   * Provides hover information for the symbol at the given position.  
   * Returns null if no symbol or no useful information available.  
   */  
  async provideHover(uri: string, position: Position): Promise<Hover | null> {  
    // TODO 1: Use languageEngine.findSymbolAt to get symbol at position  
    // TODO 2: If no symbol found, return null  
    // TODO 3: Use languageEngine.getHoverInformation to get symbol details  
    // TODO 4: Extract type information using extractTypeInformation  
    // TODO 5: Gather documentation using gatherDocumentation  
    // TODO 6: Format combined information using formatHoverContent  
    // TODO 7: Return Hover object with MarkupContent and symbol range  
    // Hint: Include source location info for symbols defined elsewhere  
    // Hint: Show function signatures with parameter names and types
```

```
}

/**
 * Formats type information and documentation into readable MarkupContent.
 */

private formatHoverContent(typeInfo: string, documentation: string): MarkupContent {

    // TODO 1: Start with code block containing type signature

    // TODO 2: Add horizontal separator if documentation exists

    // TODO 3: Add formatted documentation with proper markdown

    // TODO 4: Include usage examples if available

    // TODO 5: Add source location info if symbol is from another file

    // TODO 6: Return MarkupContent with kind='markdown'

    // Hint: Use `typescript` code blocks for syntax highlighting

    // Hint: Keep hover content concise - users scan quickly

}

}
```

```
// src/providers/definition/definition-provider.ts
```

TYPESCRIPT

```
export interface Location {
```

```
    uri: string;
```

```
    range: Range;
```

```
}
```

```
export class DefinitionProvider {
```

```
    constructor(private languageEngine: LanguageEngine) {}
```

```
/**
```

```
 * Finds definition locations for the symbol at the given position.
```

```
 * Returns array of locations (may be multiple for overloaded functions).
```

```
*/
```

```
async provideDefinition(uri: string, position: Position): Promise<Location[]> {
```

```
    // TODO 1: Use languageEngine.findSymbolAt to identify symbol at position
```

```
    // TODO 2: If no symbol found, return empty array
```

```
    // TODO 3: Use languageEngine.findDefinition to get declaration sites
```

```
    // TODO 4: For imported symbols, use resolveImportedSymbol to follow imports
```

```
    // TODO 5: Filter out invalid or inaccessible definitions
```

```
    // TODO 6: Rank definitions by relevance (interface before implementation)
```

```
    // TODO 7: Convert definition sites to Location objects
```

```
    // TODO 8: Return array of locations ordered by relevance
```

```
    // Hint: Handle special case where position is already at definition
```

```
    // Hint: For overloaded functions, return all overload definitions
```

```
}
```

```
/**
```

```
 * Resolves imported symbols to their original definition across files.
```

```
*/\n\nprivate async resolveImportedSymbol(uri: string, symbolName: string): Promise<Location | null> {\n\n    // TODO 1: Parse import statements in the current file\n\n    // TODO 2: Find import statement that brings this symbol into scope\n\n    // TODO 3: Resolve relative/absolute import path to actual file URI\n\n    // TODO 4: Use languageEngine to analyze the target file\n\n    // TODO 5: Find the exported symbol definition in target file\n\n    // TODO 6: If target file re-exports, recursively follow the chain\n\n    // TODO 7: Return the ultimate definition location\n\n    // Hint: Handle aliased imports (import { foo as bar })\n\n    // Hint: Cache import resolution results for performance\n\n}\n\n}
```

```
// src/providers/diagnostics/diagnostic-provider.ts
```

TYPESCRIPT

```
export interface Diagnostic {  
  
    range: Range;  
  
    severity?: number;  
  
    message: string;  
  
    source?: string;  
  
    code?: string | number;  
  
}
```

```
export class DiagnosticProvider {
```

```
    constructor(  
  
        private languageEngine: LanguageEngine,  
  
        private sendNotification: (method: string, params: any) => void  
    ) {}
```

```
/**
```

```
* Analyzes document and publishes diagnostics to the client.
```

```
* Runs multiple analysis phases with different priorities.
```

```
*/
```

```
async publishDiagnostics(uri: string, document: TextDocumentItem): Promise<void> {
```

```
    // TODO 1: Start with fast syntax analysis using analyzeSyntax
```

```
    // TODO 2: Publish syntax diagnostics immediately for quick feedback
```

```
    // TODO 3: If syntax analysis succeeds, run analyzeSemantics
```

```
    // TODO 4: Merge syntax and semantic diagnostics
```

```
    // TODO 5: Publish combined diagnostics to client
```

```
    // TODO 6: Start background style analysis using analyzeStyle
```

```
    // TODO 7: When style analysis completes, merge and publish final diagnostics
```

```
    // Hint: Use sendNotification('textDocument/publishDiagnostics', { uri, diagnostics })
```

```
// Hint: Clear previous diagnostics by publishing empty array first
}

/**
 * Performs fast syntax validation returning structural errors.
 */
private analyzeSyntax(uri: string, text: string): Diagnostic[] {

    // TODO 1: Use languageEngine to parse document into AST

    // TODO 2: Collect any parse errors from the parsing process

    // TODO 3: Convert parse errors to Diagnostic objects with Error severity

    // TODO 4: Check for basic structural issues (unmatched brackets, etc.)

    // TODO 5: Return array of syntax diagnostics

    // Hint: Parse errors prevent further analysis, so report them as severity 1 (Error)

    // Hint: Include helpful error messages with suggestions when possible

}

/**
 * Performs semantic validation using symbol resolution and type checking.
 */
private async analyzeSemantics(uri: string, ast: ASTNode): Promise<Diagnostic[]> {

    // TODO 1: Use languageEngine to build symbol table from AST

    // TODO 2: Check for undefined variable references

    // TODO 3: Validate function call signatures against declarations

    // TODO 4: Check type compatibility in assignments and expressions

    // TODO 5: Verify imported symbols exist and are accessible

    // TODO 6: Convert semantic errors to Diagnostic objects

    // TODO 7: Return array of semantic diagnostics with appropriate severity
}
```

```
// Hint: Type errors are usually Error severity, missing imports are Warning  
  
// Hint: Include related information pointing to relevant declarations  
  
}  
  
}
```

Milestone Checkpoints:

After Milestone 3 (Language Features):

- Run test: `npm test src/providers/completion/` - should show completion items for identifiers in scope
- Manual test: Open a file, type a partial identifier, request completion - should see filtered suggestions
- Manual test: Hover over a function name - should see signature and documentation
- Manual test: Click go-to-definition on a variable - should jump to declaration
- Expected: Completion popup appears within 200ms, hover information displays immediately, definition navigation works across files

After Milestone 4 (Diagnostics & Code Actions):

- Run test: `npm test src/providers/diagnostics/` - should detect syntax and semantic errors
- Manual test: Introduce a syntax error - should see red squiggly lines within 1 second
- Manual test: Reference an undefined variable - should see error diagnostic with helpful message
- Manual test: Use deprecated API - should see warning diagnostic with suggestion
- Expected: Error diagnostics appear immediately after typing, warnings appear within 2-3 seconds

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Completion returns empty list	Symbol resolution failing or scope walking broken	Add logging to <code>getSymbolsInScope</code> to see what symbols are found	Check AST node identification and scope hierarchy traversal
Hover shows wrong information	Position conversion error or stale cache	Log the actual position coordinates and resolved symbol	Verify UTF-16 position conversion and cache invalidation
Go-to-definition jumps to wrong location	Import resolution failing or multiple definitions not ranked	Log the definition resolution process and ranking	Check import path resolution and definition ranking logic
Diagnostics don't appear	Publishing mechanism broken or analysis throwing exceptions	Check client receives <code>textDocument/publishDiagnostics</code> notifications	Add error handling around analysis phases and ensure notifications are sent
Performance degradation	Lack of caching or inefficient symbol resolution	Profile the completion/hover request timing	Add caching for expensive operations and optimize symbol table lookups

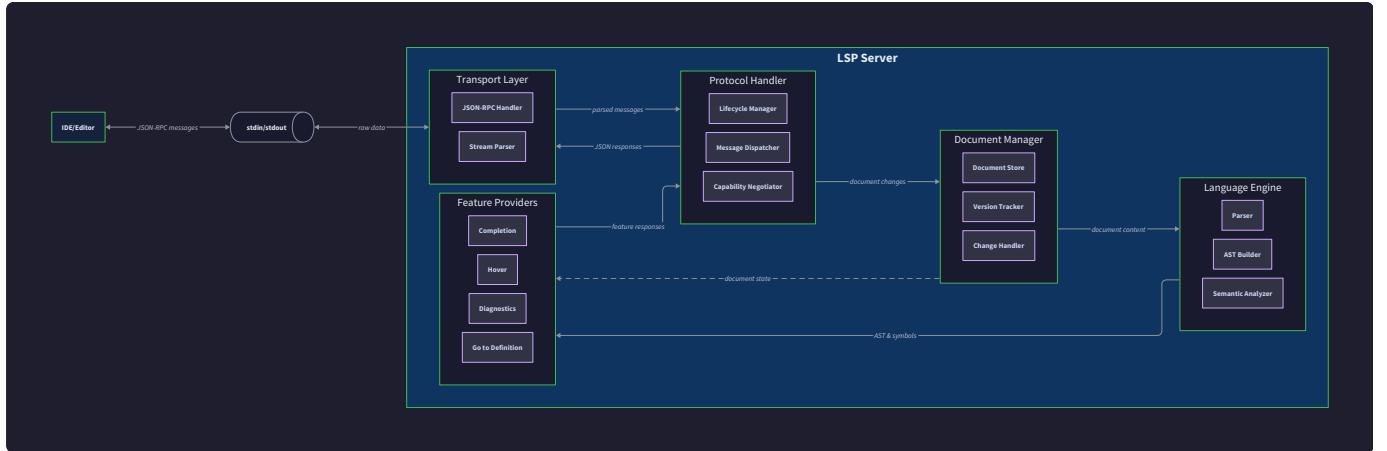
Interactions and Data Flow

Milestone(s): This section spans all milestones, with particular emphasis on Milestone 1 (JSON-RPC & Initialization) for request-response patterns, Milestone 2 (Document Synchronization) for document change flows, Milestone 3 (Language Features) for feature request sequences, and Milestone 4 (Diagnostics & Code Actions) for diagnostic publishing patterns.

Understanding how components collaborate within an LSP server is like understanding the choreography of a complex dance performance. Each component has specific movements and timing, but the magic happens in how they coordinate their actions to create a seamless experience. The LSP server orchestrates multiple concurrent conversations with the language client while maintaining consistent internal state and providing responsive language analysis.

This section examines two critical interaction patterns that define the LSP server's behavior: the synchronous request-response flow for language features and the asynchronous document change flow for maintaining

analysis state. These patterns represent the fundamental rhythms of LSP operation - the quick back-and-forth of feature requests and the continuous background work of keeping language analysis current with editor changes.



Request-Response Flow

Think of the request-response flow as a well-orchestrated relay race where each component receives the baton (request data), performs its specialized task, and passes refined information to the next component. The `LspServer` acts as the race coordinator, ensuring each handoff happens smoothly and tracking overall progress toward the finish line (client response).

The request-response flow represents the synchronous interaction pattern where the language client sends a request and expects a timely response with specific language information. This flow handles requests like `textDocument/completion`, `textDocument/hover`, `textDocument/definition`, and `textDocument/codeAction`. Each request follows a consistent journey through the server architecture, but with specialized processing based on the requested feature.

Message Reception and Routing

The flow begins when the `StreamTransport` receives data from `stdin`. The transport layer doesn't immediately know whether incoming bytes represent a complete message, partial message, or multiple messages concatenated together. The `MessageFramer` accumulates bytes in an internal buffer and repeatedly calls `tryExtractMessage()` to detect complete JSON-RPC messages based on `Content-Length` headers.

Processing Stage	Component	Responsibility	Error Handling
Data Reception	StreamTransport	Read bytes from stdin, handle partial reads	Retry on EAGAIN, report connection errors
Message Framing	MessageFramer	Parse Content-Length, extract complete messages	Validate header format, handle malformed frames
JSON Parsing	MessageFramer	Parse JSON-RPC structure	Return parse error response with error code
Message Classification	MessageDispatcher	Identify request vs notification vs response	Route based on presence of id field
Method Routing	MessageDispatcher	Route to registered handler	Return METHOD_NOT_FOUND error

Once the `MessageFramer` successfully extracts a complete message, it passes the parsed `JsonRpcMessage` to the `MessageDispatcher`. The dispatcher examines the message structure to classify it as a request (has `id` and `method`), notification (has `method` but no `id`), or response (has `id` but no `method`). For requests, the dispatcher looks up the registered handler for the specified method and invokes it with the request parameters.

Protocol State Validation

Before processing any language feature request, the `LspServer` validates that the server is in the appropriate protocol state. Most language features require the server to be in the `Initialized` state, having completed the initialization handshake. Requests received before initialization should return an error response indicating the server is not ready.

Server State	Allowed Requests	Prohibited Requests	Error Response
Uninitialized	initialize	All language features	Server not initialized
Initializing	None	All requests	Server initializing
Initialized	All language features	initialize	Invalid request
Shutdown	None	All requests	Server shutting down

Document Retrieval and Validation

For language feature requests that operate on specific documents, the request handler extracts the document URI from the request parameters and calls `getDocument(uri)` on the `DocumentManager`. This retrieval includes validation that the document exists in the server's memory and that the requested position falls within the document bounds.

The position validation is critical because LSP positions use UTF-16 code units, which can create mismatches if the client and server interpret character boundaries differently. The `DocumentManager` provides `positionToOffset()` and `offsetToPosition()` methods that handle UTF-16 encoding correctly, including proper handling of surrogate pairs for characters outside the Basic Multilingual Plane.

Position Validation Algorithm:

1. Retrieve document text from `DocumentManager`
2. Validate line number is within document bounds (0 to line count - 1)
3. Convert LSP position to byte offset using `positionToOffset()`
4. Validate byte offset falls within document text length
5. Return validation error if position is invalid
6. Proceed with feature request if position is valid

Language Analysis Coordination

Once document and position validation succeeds, the request handler delegates to the appropriate feature provider, which coordinates with the `LanguageEngine` to perform the necessary analysis. This coordination follows a consistent pattern regardless of the specific feature being requested.

The `LanguageEngine` first ensures it has current analysis information for the requested document. If the document has changed since the last analysis, the engine triggers re-parsing and symbol table reconstruction. This ensures that language features always operate on current information, even if document changes arrived shortly before the request.

Analysis Phase	LanguageEngine Method	Purpose	Caching Strategy
Parse Validation	<code>analyzeDocument()</code>	Ensure current AST exists	Invalidate on document change
Symbol Resolution	<code>buildSymbolTable()</code>	Ensure current symbol information	Incremental update when possible
Context Analysis	<code>findSymbolAt()</code>	Identify symbol at request position	Position-based lookup
Scope Resolution	Internal scope walking	Determine visible symbols	Walk up scope hierarchy

Feature-Specific Processing

Each feature provider implements its specialized logic using the analysis information from the `LanguageEngine`. The processing differs significantly between features but follows consistent patterns for error handling and response formatting.

For completion requests, the `CompletionProvider` calls `getCompletionItems()` to retrieve context-aware suggestions. The method analyzes the cursor position to determine the completion context (inside function call, after dot operator, at statement level) and filters the available symbols accordingly. The provider then formats the results as `CompletionItem` objects with appropriate labels, kinds, and documentation.

Feature	Primary Method	Context Analysis	Response Format
Completion	<code>getCompletionItems()</code>	Cursor position, partial identifier	Array of <code>CompletionItem</code>
Hover	<code>getHoverInformation()</code>	Symbol at position	<code>Hover</code> with <code>MarkupContent</code>
Definition	<code>findDefinition()</code>	Symbol reference resolution	Array of <code>Location</code>
References	<code>findReferences()</code>	Symbol usage across documents	Array of <code>Location</code>
Code Actions	<code>getCodeActions()</code>	Diagnostics and selection range	Array of code action objects

Response Construction and Serialization

After the feature provider completes its analysis, the request handler constructs the appropriate LSP response structure. This involves formatting the analysis results according to the LSP specification and ensuring all required fields are present with correct types.

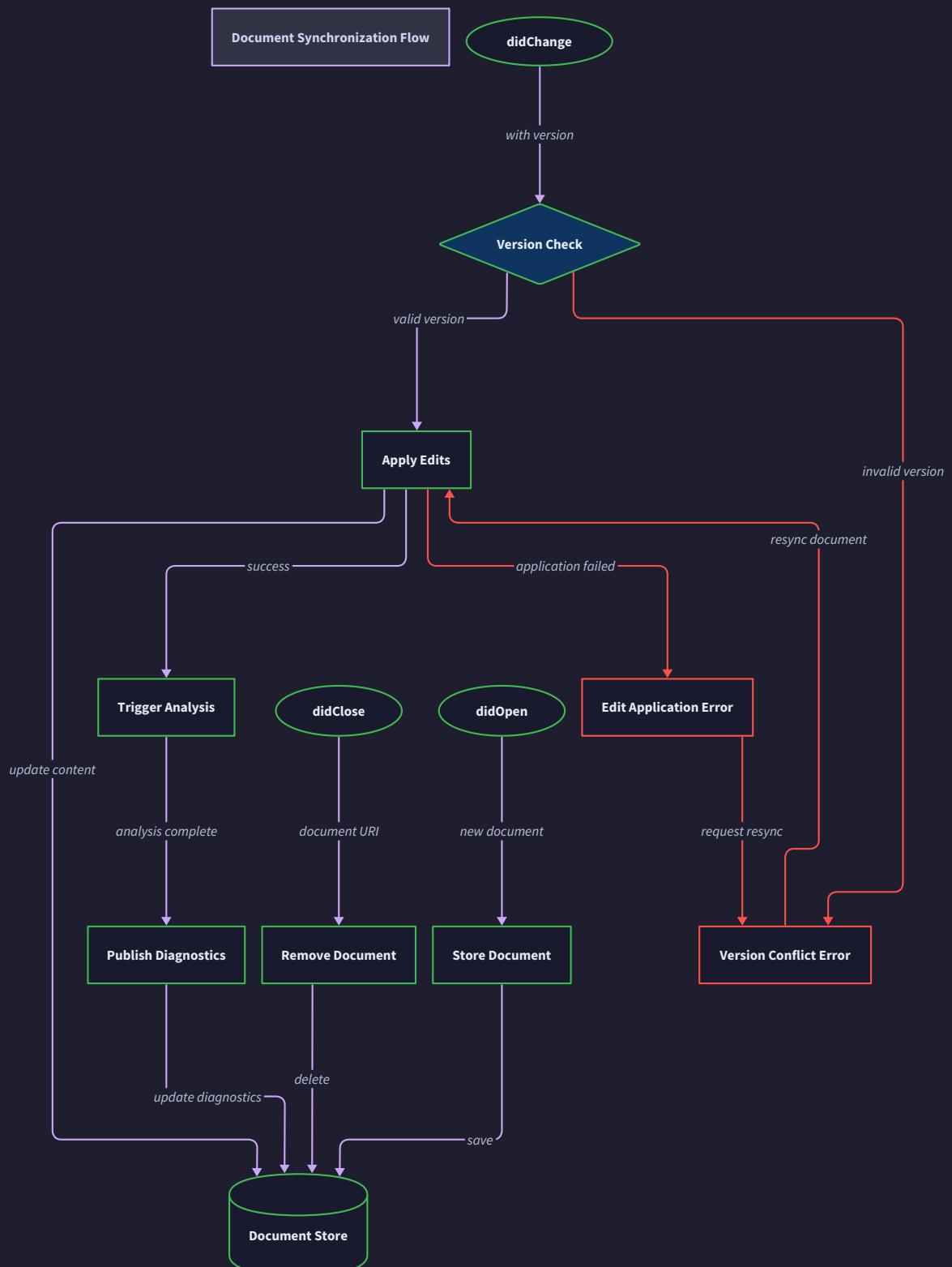
The response construction includes important details like converting internal position representations back to LSP positions, ensuring URI format consistency, and providing appropriate fallback values for optional fields. For example, hover responses must include `MarkupContent` with a specified kind (`plaintext` or `markdown`) and properly escaped content.

Once the response object is constructed, the `MessageDispatcher` calls `sendResponse()` to serialize the response as a JSON-RPC message and send it back to the client via the `StreamTransport`. The serialization includes adding the `jsonrpc: "2.0"` field and copying the request `id` to ensure proper correlation.

Error Response Handling

When errors occur during request processing, the server must return properly formatted JSON-RPC error responses rather than allowing exceptions to propagate. The error response includes a standard error code, descriptive message, and optional additional data providing context about the failure.

Error Category	JSON-RPC Code	When It Occurs	Client Behavior
Parse Error	-32700	Invalid JSON in request	Show parsing error message
Invalid Request	-32600	Malformed JSON-RPC structure	Show protocol error message
Method Not Found	-32601	Unrecognized method name	Disable unsupported feature
Invalid Params	-32602	Missing or wrong parameter types	Show parameter validation error
Internal Error	-32603	Server-side processing error	Show generic server error



Document Change Flow

The document change flow operates like a newsroom's editorial process - when breaking news arrives (document changes), it triggers a coordinated response involving fact-checking (parsing), analysis (semantic analysis), and publication (diagnostic updates). Unlike the request-response flow's synchronous nature, document changes initiate an asynchronous cascade of analysis and notification activities.

Document changes represent one of the most performance-critical aspects of LSP server design because they can arrive rapidly during active editing sessions. A single keystroke might trigger document change notifications, which in turn trigger parsing, symbol analysis, and diagnostic computation. The server must process these changes efficiently while maintaining analysis accuracy and avoiding overwhelming the client with excessive diagnostic updates.

Document Change Detection and Buffering

Document changes arrive as `textDocument/didChange` notifications containing the document URI, new version number, and an array of `TextDocumentContentChangeEvent` objects. Each change event specifies either a full document replacement (no `range` field) or an incremental change with a specific range and replacement text.

The `DocumentManager` processes these notifications through the `handleDidChange()` method, which first validates the version number to ensure changes are applied in the correct order. Version numbers must increase monotonically for each document, and the server should reject changes with version numbers that are out of sequence.

Change Event Type	Range Field	RangeLength Field	Text Field	Processing Method
Full Replacement	undefined	undefined	Complete document text	Replace entire document content
Incremental Change	Start and end positions	Optional character count	Replacement text	Apply range-based edit
Insert	Zero-length range	0	New text to insert	Insert text at position
Delete	Range to remove	Character count	Empty string	Remove text in range

Incremental Change Application

Incremental changes require careful position arithmetic to avoid corrupting the document content. The `DocumentManager` converts LSP positions to byte offsets using `positionToOffset()`, applies the text changes, and then recalculates positions for any subsequent changes in the same batch.

The critical challenge is that applying one change can invalidate the positions of subsequent changes in the same batch. For example, if the first change inserts text at line 5, then a second change targeting line 10 must account for the position shift caused by the insertion. The server must apply changes in the correct order and adjust positions dynamically.

Incremental Change Algorithm:

1. Sort changes by position (earliest first) to ensure correct application order
2. For each change in the sorted list:
 - a. Convert LSP range to byte offsets using current document state
 - b. Validate that range falls within current document bounds
 - c. Extract the text to be replaced for validation
 - d. Apply the replacement text at the calculated offset
 - e. Update document length and line count
3. Increment document version number
4. Update document timestamp for cache invalidation
5. Notify change listeners of the document update

Change Listener Notification

After successfully applying document changes, the `DocumentManager` notifies registered change listeners about the update. This notification system allows other components to respond to document changes without tight coupling. The `LanguageEngine` and `DiagnosticProvider` register as change listeners to trigger analysis when documents are modified.

The notification includes the document URI, new version number, and the nature of the changes (full replacement vs incremental updates). Listeners can use this information to make intelligent decisions about cache invalidation and analysis scheduling. For example, small incremental changes might trigger incremental parsing, while large changes might require full re-parsing.

Analysis Triggering and Scheduling

When the `LanguageEngine` receives a document change notification, it must decide whether to trigger immediate analysis or schedule analysis for later execution. This decision involves balancing responsiveness (users want quick feedback) against performance (parsing large files is expensive) and avoiding analysis thrashing during rapid typing.

The engine typically implements a debouncing strategy where analysis is delayed for a short period (e.g., 500 milliseconds) after each change. If additional changes arrive during the delay period, the timer resets. This ensures that analysis occurs only after the user pauses typing, reducing computational overhead during active editing sessions.

Change Frequency	Analysis Strategy	Delay Duration	Resource Impact
Single change	Immediate analysis	0ms	Low
Rapid changes (< 500ms apart)	Debounced analysis	500ms from last change	Medium
Continuous typing	Analysis suspended	Until typing pause	High
Large file changes	Background analysis	Immediate, lower priority	Variable

Parse Tree Invalidation and Reconstruction

When analysis begins, the `LanguageEngine` first invalidates cached analysis information for the changed document. This includes the abstract syntax tree, symbol tables, and any derived analysis results like type information or error diagnostics. The invalidation must be precise to avoid unnecessary work while ensuring correctness.

For incremental changes, the engine attempts to determine which portions of the analysis can be preserved. If changes occur within a single function body, the engine might preserve the file-level AST structure and re-parse only the affected function. However, changes to declarations, imports, or global scope require more comprehensive re-analysis.

The engine calls `analyzeDocument()` to reconstruct the parse tree and symbol information. This method parses the updated document text, constructs a new AST, and rebuilds the symbol table by walking the tree structure. The parsing process includes syntax error detection and recovery, allowing analysis to continue even when the document contains incomplete or malformed code.

Symbol Table Updates and Cross-File Impact

Symbol table reconstruction is more complex than parsing because symbol changes can impact other files through import relationships. When a document change modifies exported symbols (functions, types, constants), the engine must identify and update other files that import those symbols.

The engine maintains a dependency graph tracking which files import symbols from other files. When a file's exported symbols change, the engine schedules re-analysis for all dependent files. This ensures that cross-file features like go-to-definition and completion remain accurate after changes.

Symbol Change Type	Local Impact	Cross-File Impact	Analysis Scope
Local variable	Single function	None	Function-level
Function signature	Single file	All importers	File + dependents
Type definition	Single file	All importers + implementers	File + dependents
Export/import changes	Single file	All dependents in dependency chain	Potentially large scope

Diagnostic Computation and Publishing

After completing symbol table updates, the engine calls `getDiagnostics()` to identify syntax and semantic errors in the updated document. Diagnostic computation includes syntax validation (malformed code structures), semantic validation (type errors, undefined references), and style warnings (unused variables, deprecated usage).

The diagnostic computation process must handle partial or invalid code gracefully. During active editing, documents frequently contain incomplete statements or syntax errors. The parser includes error recovery logic that attempts to construct meaningful AST structures even from invalid code, allowing semantic analysis to provide useful diagnostics and features.

Once diagnostic computation completes, the server publishes diagnostics to the client using the `textDocument/publishDiagnostics` notification. This notification includes the document URI, version

number, and an array of `Diagnostic` objects describing each identified issue.

Diagnostic Publishing Algorithm:

1. Complete parsing and semantic analysis for the changed document
2. Collect all syntax errors from the parser
3. Collect semantic errors from symbol resolution and type checking
4. Format errors as `Diagnostic` objects with ranges, severity, and messages
5. Send `publishDiagnostics` notification to client with diagnostic array
6. Update internal diagnostic cache for consistency
7. If cross-file analysis identified issues, repeat for affected files

Performance Optimization and Throttling

The document change flow includes several optimization strategies to maintain responsiveness under heavy editing loads. These optimizations balance analysis accuracy against performance, ensuring the server remains responsive even when processing large files or rapid change sequences.

The engine implements a multi-tier caching strategy where frequently accessed analysis results are kept in fast memory, while less frequently used results are moved to slower storage or recomputed on demand. Cache entries include access timestamps and usage counts to support intelligent eviction policies.

Optimization Strategy	Implementation	Performance Impact	Accuracy Trade-off
Change debouncing	Timer-based delay	Reduces analysis frequency	Delayed feedback
Incremental parsing	AST node preservation	Faster re-analysis	Complex invalidation logic
Analysis caching	Multi-tier cache with LRU eviction	Faster repeated analysis	Memory usage
Background analysis	Lower-priority threads	Non-blocking UI updates	Potentially stale results
Throttled diagnostics	Rate-limited publishing	Reduced client message load	Diagnostic delays

Design Insight: The asynchronous nature of document changes creates a fundamental tension between immediate feedback and system performance. The most successful LSP servers implement sophisticated scheduling and caching strategies that provide rapid feedback for common editing patterns while gracefully degrading performance for complex scenarios.

Error Recovery and Consistency Maintenance

The document change flow must handle various error conditions without corrupting the server's internal state or losing synchronization with the client. Error recovery includes handling invalid change ranges, version number mismatches, and analysis failures due to resource constraints.

When document change application fails, the server should attempt to resynchronize with the client by requesting a full document update. This recovery mechanism ensures that temporary communication errors don't permanently desynchronize the client and server document states.

Error Condition	Detection Method	Recovery Action	Prevention Strategy
Invalid change range	Position validation	Request full document sync	Robust position conversion
Version mismatch	Version number comparison	Log warning, apply anyway	Strict version tracking
Parse failure	Exception handling	Preserve previous AST	Error recovery in parser
Memory exhaustion	Resource monitoring	Skip analysis, log error	Analysis result caching
Analysis timeout	Timer-based limits	Return partial results	Incremental analysis

Implementation Guidance

The implementation of interaction flows requires careful coordination between multiple concurrent processes and robust error handling to maintain consistency under various failure scenarios.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Request Routing	Map-based method dispatch	Event-driven architecture with queues
Change Debouncing	setTimeout-based delays	Priority queue with backoff strategies
Analysis Scheduling	Immediate execution	Worker thread pool with priority levels
Cache Management	Simple Map with size limits	LRU cache with memory pressure handling
Error Recovery	Try-catch with logging	Circuit breaker pattern with fallback modes

B. Recommended File/Module Structure:

```

src/
  server/
    lsp-server.ts           ← main LspServer class
    message-dispatcher.ts   ← request routing and response handling
  flows/
    request-handler.ts     ← request-response flow coordination
    document-sync-handler.ts ← document change flow coordination
  providers/
    completion-provider.ts ← completion request implementation
    hover-provider.ts       ← hover request implementation
    diagnostic-provider.ts  ← diagnostic publishing implementation
  utils/
    debouncer.ts            ← change debouncing utility
    position-utils.ts        ← position conversion helpers
  
```

C. Infrastructure Starter Code (COMPLETE, ready to use):

Change Debouncing Utility (`utils/debouncer.ts`):

```
export class ChangeDebouncer {
```

TYPESCRIPT

```
    private timers: Map<string, NodeJS.Timeout> = new Map();
```

```
    debounce(key: string, delay: number, callback: () => void): void {
```

```
        // Clear existing timer for this key
```

```
        const existingTimer = this.timers.get(key);
```

```
        if (existingTimer) {
```

```
            clearTimeout(existingTimer);
```

```
}
```

```
        // Set new timer
```

```
        const timer = setTimeout(() => {
```

```
            this.timers.delete(key);
```

```
            callback();
```

```
        }, delay);
```

```
        this.timers.set(key, timer);
```

```
}
```

```
    cancel(key: string): void {
```

```
        const timer = this.timers.get(key);
```

```
        if (timer) {
```

```
            clearTimeout(timer);
```

```
            this.timers.delete(key);
```

```
}
```

```
}
```

```
cancelAll(): void {  
  
    for (const timer of this.timers.values()) {  
  
        clearTimeout(timer);  
  
    }  
  
    this.timers.clear();  
  
}  
  
}
```

Position Conversion Utilities (`utils/position-utils.ts`):

```
import { Position, Range } from '../types';
```

TYPESCRIPT

```
export function positionToOffset(text: string, position: Position): number {
```

```
    const lines = text.split('\n');
```

```
    if (position.line >= lines.length) {
```

```
        return text.length;
```

```
}
```

```
    let offset = 0;
```

```
    for (let i = 0; i < position.line; i++) {
```

```
        offset += lines[i].length + 1; // +1 for newline character
```

```
}
```

```
    const lineText = lines[position.line];
```

```
    const character = Math.min(position.character, lineText.length);
```

```
    return offset + character;
```

```
}
```

```
export function offsetToPosition(text: string, offset: number): Position {
```

```
    const lines = text.split('\n');
```

```
    let currentOffset = 0;
```

```
    for (let line = 0; line < lines.length; line++) {
```

```
        const lineLength = lines[line].length;
```

```
        if (currentOffset + lineLength >= offset) {
```

```
            return { line, character: offset - currentOffset };
```

```
}
```

```
        currentOffset += lineLength + 1; // +1 for newline
```

```
}

// Offset beyond end of document

return { line: lines.length - 1, character: lines[lines.length - 1].length };

}

export function rangeContainsPosition(range: Range, position: Position): boolean {

if (position.line < range.start.line || position.line > range.end.line) {

    return false;

}

if (position.line === range.start.line && position.character < range.start.character) {

    return false;

}

if (position.line === range.end.line && position.character > range.end.character) {

    return false;

}

return true;

}
```

D. Core Logic Skeleton Code (signature + TODOs only):

Request Handler (`flows/request-handler.ts`):

```
import { JsonRpcRequest, JsonRpcResponse } from '../types';
```

TYPESCRIPT

```
import { DocumentManager } from '../document/document-manager';
```

```
import { LanguageEngine } from '../engine/language-engine';
```

```
export class RequestHandler {
```

```
    constructor(
```

```
        private documentManager: DocumentManager,
```

```
        private languageEngine: LanguageEngine
```

```
    ) {}
```

```
    /**
     * Processes a language feature request and returns the appropriate response.
     * Coordinates document validation, analysis, and feature-specific processing.
     */

```

```
    async handleLanguageFeatureRequest(request: JsonRpcRequest): Promise<JsonRpcResponse> {
```

```
        // TODO 1: Extract document URI and position from request parameters
```

```
        // TODO 2: Call validateDocumentAndPosition() to ensure request is valid
```

```
        // TODO 3: Ensure language analysis is current for the document
```

```
        // TODO 4: Route to appropriate feature provider based on request method
```

```
        // TODO 5: Format provider response according to LSP specification
```

```
        // TODO 6: Return JsonRpcResponse with result or error
```

```
        // Hint: Use switch statement on request.method for routing
```

```
}
```

```
    /**
     * Validates that the requested document exists and position is within bounds.
     * Returns error details if validation fails, null if successful.
     */

```

```

private validateDocumentAndPosition(uri: string, position: Position): string | null {

    // TODO 1: Call documentManager.getDocument(uri) to retrieve document

    // TODO 2: Return error message if document not found

    // TODO 3: Call positionToOffset() to validate position is within document

    // TODO 4: Return error message if position is invalid

    // TODO 5: Return null if validation succeeds

    // Hint: Check both line number bounds and character bounds

}

/**
 * Ensures the language engine has current analysis for the specified document.
 * Triggers re-analysis if the document has changed since last analysis.
 */
private async ensureCurrentAnalysis(uri: string): Promise<void> {

    // TODO 1: Get document from DocumentManager

    // TODO 2: Check if LanguageEngine analysis is stale

    // TODO 3: Call languageEngine.analyzeDocument() if re-analysis needed

    // TODO 4: Wait for analysis completion before returning

    // Hint: Compare document version with last analyzed version

}

```

Document Synchronization Handler (`flows/document-sync-handler.ts`):

```
import { DidChangeTextDocumentParams, TextDocumentContentChangeEvent } from '../types';  
import { DocumentManager } from '../document/document-manager';  
import { LanguageEngine } from '../engine/language-engine';  
import { ChangeDebouncer } from '../utils/debouncer';  
  
export class DocumentSyncHandler {  
  
    private debouncer = new ChangeDebouncer();  
  
    constructor(  
        private documentManager: DocumentManager,  
        private languageEngine: LanguageEngine,  
        private diagnosticPublisher: (uri: string, diagnostics: Diagnostic[]) => void  
    ) {}  
  
    /**  
     * Handles document change notifications and triggers analysis cascade.  
     * Applies changes, schedules debounced analysis, and publishes diagnostics.  
     */  
  
    async handleDidChange(params: DidChangeTextDocumentParams): Promise<void> {  
        // TODO 1: Extract document URI and version from params  
        // TODO 2: Call documentManager.applyChanges() with change events  
        // TODO 3: Cancel existing debounced analysis for this document  
        // TODO 4: Schedule new debounced analysis using this.debouncer  
        // TODO 5: In debounced callback, trigger analysis and diagnostic publishing  
        // Hint: Use 500ms delay for debouncing, key by document URI  
    }  
  
    /**
```

```

    * Applies an array of content changes to the document in correct order.

    * Handles both full replacement and incremental changes.

    */

private applyContentChanges(
    uri: string,
    version: number,
    changes: TextDocumentContentChangeEvent[]
): void {

    // TODO 1: Sort changes by position (earliest first) for correct application

    // TODO 2: Get current document text from DocumentManager

    // TODO 3: For each change, determine if it's full replacement or incremental

    // TODO 4: For incremental changes, convert positions to offsets and apply

    // TODO 5: Update document with final text and new version number

    // Hint: Changes without range field are full replacements

}

/** 

 * Triggers language analysis and publishes resulting diagnostics.

 * Handles analysis errors gracefully and publishes empty diagnostics on failure.

 */

private async analyzeAndPublishDiagnostics(uri: string): Promise<void> {

    // TODO 1: Get document from DocumentManager

    // TODO 2: Call languageEngine.analyzeDocument() to trigger analysis

    // TODO 3: Call languageEngine.getDiagnostics() to get error list

    // TODO 4: Call diagnosticPublisher with URI and diagnostic array

    // TODO 5: Handle analysis errors by publishing empty diagnostic array

    // Hint: Wrap analysis calls in try-catch for error recovery
}

```

```
    }  
  
}
```

E. Language-Specific Hints:

- Use `setTimeout` and `clearTimeout` for implementing change debouncing in TypeScript
- The `Map` data structure provides efficient key-value storage for tracking pending analysis operations
- Use `Promise.resolve()` and `async/await` for coordinating asynchronous analysis operations
- Array's `sort()` method with custom comparator can order document changes by position
- String's `split('\n')` method helps convert between line/character and offset positions
- Use union types like `JsonRpcRequest | JsonRpcNotification` for message discrimination

F. Milestone Checkpoint:

After implementing the interaction flows, verify correct behavior:

Request-Response Flow Testing:

```
# Start your LSP server  
  
npm start  
  
# In another terminal, send a completion request  
  
echo -e "Content-Length:  
168\r\n\r\n{\\"jsonrpc\\":\"2.0\",\\\"id\\\":1,\\\"method\\\":\\\"textDocument/completion\\\",\\\"params\\\":  
{\\\"textDocument\\\":{\\\"uri\\\":\\\"file:///test.js\\\"},\\\"position\\\":{\\\"line\\\":0,\\\"character\\\":10}\\}}}"  
| node dist/server.js
```

Expected behavior: Server should respond with completion items or appropriate error message. Check that:

- Request routing works correctly based on method name
- Document validation prevents requests for non-existent documents
- Position validation prevents out-of-bounds requests
- Feature providers receive correctly formatted parameters

Document Change Flow Testing:

1. Open a document with `textDocument/didOpen`
2. Send rapid `textDocument/didChange` notifications (within 500ms of each other)
3. Verify that analysis is debounced and occurs only after changes stop
4. Check that `textDocument/publishDiagnostics` notifications are sent with current diagnostics

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Completion requests timeout	Analysis blocking main thread	Add logging to analysis methods	Implement analysis scheduling with setTimeout
Stale completion results	Document changes not triggering re-analysis	Check change listener registration	Ensure DocumentManager notifies LanguageEngine of changes
Excessive diagnostic notifications	Missing debouncing in change flow	Count publishDiagnostics messages	Implement proper change debouncing with timer reset
Invalid position errors	Position conversion bugs	Log position values and document bounds	Verify UTF-16 handling in positionToOffset
Memory leaks during editing	Unbounded cache growth	Monitor memory usage over time	Implement cache size limits and LRU eviction

Error Handling and Edge Cases

Milestone(s): This section spans all milestones but has particular relevance to Milestone 1 (JSON-RPC & Initialization) for protocol error handling, Milestone 2 (Document Synchronization) for malformed change events, Milestone 3 (Language Features) for parse error recovery, and Milestone 4 (Diagnostics & Code Actions) for graceful degradation under analysis failures.

Building a robust Language Server Protocol implementation requires comprehensive error handling at multiple levels of the system. Think of error handling in an LSP server like building a multi-layered safety net for a complex juggling act. The language client is constantly throwing balls (JSON-RPC messages, document changes, feature requests) at our server, and we must catch every one gracefully, even when some balls are malformed, arrive out of order, or contain impossible requests. Unlike a simple request-response web service that can return a 500 error and move on, an LSP server maintains persistent state and ongoing relationships with documents and symbols. A single unhandled error can corrupt this state, break the communication protocol, or leave the editor in an unusable condition.

The challenge is that errors can occur at multiple layers simultaneously. A malformed JSON-RPC message might contain a valid language feature request for a document that has parse errors, while the client connection is experiencing intermittent failures. Our error handling strategy must isolate failures, maintain system coherence, provide meaningful feedback to both the language client and end user, and enable recovery wherever possible.

This section explores the two primary categories of error scenarios: protocol-level failures that affect communication between client and server, and language-level failures that occur during code analysis and feature provision.

Protocol-Level Error Handling

Protocol-level errors occur in the communication layer between the language client and server, affecting the JSON-RPC message exchange and transport mechanisms. Think of this layer as the telephone system connecting two offices - when the phone lines have static, drop calls, or deliver garbled messages, both parties need strategies to detect problems, request clarification, and re-establish clear communication.

The LSP specification defines several categories of protocol errors, each requiring different detection and recovery strategies. JSON-RPC errors have standardized error codes that clients expect, while transport-level failures like broken pipes or encoding issues require more fundamental recovery mechanisms.

JSON-RPC Error Response Handling

The JSON-RPC specification defines a structured approach to error reporting that allows the language client to understand what went wrong and potentially retry or adapt its behavior. Every JSON-RPC request can result in either a successful response or an error response, and our server must generate appropriate error responses for all failure scenarios.

Decision: Structured Error Response Strategy

- **Context:** JSON-RPC errors need consistent structure for client compatibility
- **Options Considered:** Generic error messages, structured error codes, hybrid approach
- **Decision:** Use standardized JSON-RPC error codes with detailed structured data
- **Rationale:** Clients can programmatically handle known error types while still receiving human-readable details
- **Consequences:** Enables intelligent client retry logic but requires comprehensive error classification

The following table details the standard JSON-RPC error codes and their appropriate usage contexts in our LSP server:

Error Code	Error Name	Usage Context	Example Scenario	Recovery Strategy
-32700	Parse Error	Invalid JSON received	Malformed JSON in message body	Log error, send error response, continue processing
-32600	Invalid Request	Missing required JSON-RPC fields	Missing 'method' field in request	Send error response with field requirements
-32601	Method Not Found	Unsupported LSP method	Client sends unimplemented method	Send error response listing supported methods
-32602	Invalid Params	Wrong parameter structure	Missing required position parameter	Send error response with parameter schema
-32603	Internal Error	Server-side processing failure	Unhandled exception in language engine	Log stack trace, send generic error response

Our error response generation follows a consistent pattern that provides maximum information to the language client while maintaining security boundaries. Each error response includes the original request ID for correlation, a standardized error code, a human-readable message, and structured data that helps the client understand the specific failure mode.

The `MessageDispatcher` component is responsible for catching exceptions from method handlers and converting them into appropriate JSON-RPC error responses. When a handler throws an exception, the dispatcher examines the exception type and context to determine the most appropriate error code and constructs a detailed error response.

Transport Layer Failure Detection

Transport layer failures affect the fundamental communication channel between client and server. These failures are more serious than JSON-RPC errors because they can corrupt the message stream, cause message loss, or completely break the communication protocol.

Think of transport failures like problems with the physical telephone infrastructure rather than miscommunication between the parties talking. When the phone line crackles with static, both parties hear garbled sounds but know the connection is degraded. When the line goes completely dead, both parties know they need to re-establish contact through alternative means.

Our `StreamTransport` component implements several detection mechanisms for transport-layer failures:

Connection State Monitoring: The transport layer continuously monitors the stdin/stdout streams for unexpected closure, write failures, or read timeouts. When the language client terminates abnormally, the server detects the broken pipe condition and initiates graceful shutdown procedures.

Message Framing Validation: The `MessageFramer` validates Content-Length headers against actual message sizes and detects framing errors that could indicate transport corruption. When framing errors occur repeatedly, this suggests systematic encoding or buffering issues that require connection reset.

Encoding Error Detection: Since LSP messages use UTF-8 encoding but position calculations use UTF-16 code units, encoding mismatches can cause position calculation errors. The transport layer validates character encoding consistency and reports encoding errors when they occur.

The following table describes transport failure modes and their detection strategies:

Failure Mode	Detection Method	Symptoms	Recovery Action
Broken Pipe	Write failure or stdin EOF	IOException on stdout write	Log termination reason, exit gracefully
Framing Corruption	Content-Length mismatch	Partial messages or parsing failures	Request message retransmission
Encoding Mismatch	Character conversion errors	Position calculation inconsistencies	Switch to byte-offset calculations
Buffer Overflow	Memory allocation failures	Out of memory during message parsing	Implement message size limits
Stream Timeout	Read timeout on stdin	No data received within timeout period	Send ping request to verify connection

The critical insight for transport error handling is that some failures are recoverable through protocol messages, while others require process-level recovery. A malformed Content-Length header can be handled by requesting retransmission, but a broken stdout pipe means the client has disconnected and the server should terminate.

Connection Lifecycle Management

The LSP server maintains a connection state machine that tracks the current communication status with the language client. This state machine helps coordinate recovery actions and ensures that error handling doesn't interfere with normal protocol operations.

The connection lifecycle follows these states and transitions:

Current State	Event	Next State	Actions Taken
Disconnected	stdin data available	Connecting	Initialize message framer, start reading
Connecting	Valid JSON-RPC message received	Connected	Begin normal message processing
Connected	Framing error detected	Degraded	Enable enhanced error checking
Connected	Write failure on stdout	Failed	Log error, prepare for shutdown
Degraded	Multiple successful messages	Connected	Resume normal operation
Degraded	Additional errors within timeout	Failed	Declare connection unusable
Failed	Any event	Failed	Ignore events, await shutdown signal

When the connection enters the `Degraded` state, the server implements additional validation and error checking to help identify systematic issues. This might include verifying message checksums, implementing redundant framing validation, or requesting acknowledgment for critical messages.

Client Capability Degradation

When protocol errors occur during capability negotiation or feature requests, the server implements capability degradation strategies that allow continued operation with reduced functionality. Think of this like a high-definition video call that automatically reduces quality when bandwidth becomes limited - the core communication continues, but some advanced features become unavailable.

The `CapabilityNegotiator` maintains a dynamic capability set that can be reduced in response to repeated errors in specific feature areas. If completion requests consistently fail due to client compatibility issues, the server can temporarily disable completion capabilities while continuing to provide hover and go-to-definition features.

This capability degradation is communicated to the client through server capability updates or by returning "not supported" error responses for degraded features. The client can then adapt its behavior by disabling problematic features in the user interface.

Parse Error Recovery

Parse error recovery addresses failures that occur during source code analysis, when the `LanguageEngine` encounters syntax errors, semantic inconsistencies, or partially invalid code. Unlike protocol errors that affect communication, parse errors are part of the normal language server workflow - developers frequently write code with temporary syntax errors, incomplete statements, or references to not-yet-defined symbols.

Think of parse error recovery like a skilled translator working with a poorly written document. When the translator encounters a sentence that doesn't make grammatical sense, they don't abandon the entire document. Instead, they make reasonable assumptions about the intended meaning, note the problematic areas for later correction, and continue translating the rest of the document. The resulting translation may not be perfect, but it provides maximum value while clearly identifying areas that need attention.

The key insight for LSP parse error recovery is that developers expect language features to work even when their code has errors. A single missing semicolon shouldn't disable code completion for the entire file. A typo in a variable name shouldn't prevent go-to-definition from working for correctly spelled symbols in the same scope.

Syntax Error Recovery Strategies

When the language parser encounters syntax errors, it needs strategies to recover and continue parsing the rest of the document. Our `LanguageEngine` implements several recovery techniques that allow analysis to continue despite local syntax problems.

Error Production Rules: The parser includes special grammar rules that match common error patterns and provide recovery points. For example, if a function declaration is missing its closing brace, the parser can detect the start of the next top-level declaration and assume the function ended there.

Panic Mode Recovery: When the parser encounters an unexpected token, it enters panic mode and skips tokens until it finds a known synchronization point like a semicolon, closing brace, or keyword that indicates the start of a new statement or declaration.

Phrase-Level Recovery: For common errors like missing operators or delimiters, the parser can insert the missing token and continue parsing. This requires careful heuristics to avoid cascading error correction that produces misleading results.

The following table describes syntax error recovery strategies and their application contexts:

Recovery Strategy	Application Context	Example Error	Recovery Action	Limitations
Error Productions	Missing delimiters	Function without closing brace	Assume function ends at next declaration	May misattribute code to wrong scope
Panic Mode	Unexpected tokens	Random characters in expression	Skip to next statement boundary	Loses information about skipped code
Phrase-Level	Missing operators	a b instead of a + b	Insert likely missing operator	May guess wrong operator
Balanced Delimiters	Unmatched brackets	Missing closing parenthesis	Track nesting depth, auto-close	Can't handle complex nesting errors
Context Switching	Invalid constructs	Statement in type position	Switch to appropriate parsing context	May create invalid AST structure

Decision: Multi-Strategy Error Recovery

- **Context:** Single recovery strategy insufficient for diverse error patterns
- **Options Considered:** Single panic mode, error productions only, comprehensive multi-strategy
- **Decision:** Implement layered recovery with strategy prioritization
- **Rationale:** Different error types need different recovery approaches for optimal results
- **Consequences:** More complex parser implementation but significantly better error recovery

Partial AST Construction

When syntax errors prevent complete parsing of a code construct, the `LanguageEngine` builds partial AST nodes that represent the successfully parsed portions while marking error regions. This partial AST enables language features to work on the valid code while clearly identifying problematic areas.

Error Node Insertion: The AST includes special error nodes that represent unparseable code regions. These nodes contain the raw text and error information, allowing later analysis to potentially recover some semantic information.

Incomplete Node Completion: When a declaration or statement is partially parsed, the AST node is marked as incomplete but includes all successfully parsed children. This allows symbol resolution to work with the known parts while flagging the missing information.

Scope Boundary Preservation: Even when individual statements have errors, the parser attempts to preserve scope boundaries and major structural elements. This ensures that symbol resolution can still build meaningful scope chains and identify accessible declarations.

The partial AST construction follows these principles:

1. **Maximal Information Preservation:** Include every piece of successfully parsed information in the AST, even if the overall construct is invalid
2. **Clear Error Marking:** Mark all error regions and incomplete nodes so subsequent analysis can identify reliable vs. uncertain information
3. **Structural Coherence:** Maintain valid parent-child relationships and scope nesting even when individual nodes are incomplete
4. **Recovery Point Identification:** Mark locations where parsing successfully resumed after errors

Semantic Analysis with Incomplete Information

The `LanguageEngine` performs semantic analysis even when the AST contains parse errors, implementing strategies to extract maximum meaningful information while clearly identifying uncertain conclusions.

Symbol Resolution with Missing Information: When a symbol reference cannot be resolved due to incomplete declarations or missing import statements, the analyzer records the attempted resolution and provides partial information. For example, a variable reference might resolve to a partial declaration that includes the name but lacks type information.

Type Inference with Constraints: When type information is missing due to parse errors, the type inference system uses constraint propagation to derive likely types from context. These inferred types are marked as uncertain and may be updated as more information becomes available.

Scope Chain Reconstruction: Even when scope boundaries are unclear due to syntax errors, the analyzer attempts to build meaningful scope chains by using heuristics based on indentation, partial declarations, and structural patterns.

The following table describes semantic analysis strategies for incomplete information:

Information Gap	Analysis Strategy	Confidence Level	Feature Impact
Missing Type Annotation	Constraint-based inference	Medium	Completion may include uncertain types
Incomplete Declaration	Partial symbol creation	Low	Go-to-definition may jump to incomplete location
Broken Import Statement	Module resolution fallback	Low	Imported symbols unavailable for completion
Invalid Expression	Expression boundary detection	High	Features work around expression
Missing Scope Delimiter	Indentation-based scoping	Medium	Symbol visibility may be incorrect

Graceful Feature Degradation

When parse errors make certain language features unreliable or impossible, the server implements graceful degradation that maintains maximum functionality while clearly communicating limitations to the user.

Confidence-Based Filtering: Language features like code completion filter results based on confidence levels. High-confidence results from well-parsed code are presented normally, while lower-confidence results from error-adjacent code are marked or de-prioritized.

Feature-Specific Fallbacks: Each language feature implements fallback strategies appropriate to its use case. Code completion might fall back to simple text-based suggestions, while hover information might display partial type information with uncertainty indicators.

Progressive Enhancement: As parse errors are corrected through document changes, language features automatically resume full functionality. The system continuously re-evaluates feature availability as the document state improves.

Error Reporting and User Feedback

The diagnostic system coordinates parse error reporting with language feature degradation to provide clear user feedback about both problems and limitations.

Error Classification: Parse errors are classified by severity and impact on language features. Syntax errors that prevent symbol resolution are marked as high-impact, while errors that only affect local expressions are marked as low-impact.

Feature Impact Communication: Diagnostic messages include information about which language features may be affected by each error. This helps developers prioritize which errors to fix first based on their current workflow needs.

Recovery Suggestions: Where possible, diagnostic messages include suggestions for common error corrections that would restore full language feature functionality.

⚠ Pitfall: Cascading Error Recovery A common mistake is allowing error recovery in one part of the parse to trigger additional error recovery attempts in dependent code. This can lead to a cascade of incorrect assumptions that produces an AST that's structurally valid but semantically meaningless. For example, if the parser incorrectly recovers from a missing function parameter by assuming it's a void parameter, this might make the function body appear to have invalid variable references, triggering additional incorrect recovery attempts. To prevent this, limit error recovery to local contexts and mark all recovered regions as uncertain, preventing them from influencing recovery decisions in other parts of the code.

Implementation Guidance

This section provides concrete implementation strategies for building robust error handling into your LSP server, with complete code examples for infrastructure components and detailed guidance for implementing error recovery logic.

Technology Recommendations

Component	Simple Option	Advanced Option
Error Classification	Basic exception types with error codes	Structured error hierarchy with context data
Parse Recovery	Single panic-mode recovery	Multi-strategy recovery with confidence scoring
Transport Monitoring	Basic stream status checks	Health monitoring with metrics collection
Diagnostic Reporting	Simple error list	Rich diagnostics with related information
State Management	In-memory error flags	Persistent error state with recovery tracking

Recommended File Structure

```
project-root/
  src/
    transport/
      stream-transport.ts          ← transport error handling
      message-framer.ts           ← framing validation
      connection-monitor.ts       ← connection health tracking
    protocol/
      error-handler.ts            ← JSON-RPC error responses
      protocol-state.ts           ← connection state machine
      error-recovery.ts           ← protocol recovery strategies
    language/
      parser-recovery.ts          ← syntax error recovery
      partial-ast.ts               ← incomplete AST handling
      semantic-analyzer.ts        ← analysis with missing info
    diagnostics/
      error-classifier.ts          ← error categorization
      diagnostic-publisher.ts     ← error reporting coordination
    common/
      error-types.ts               ← error type definitions
      recovery-strategies.ts       ← reusable recovery patterns
```

JSON-RPC Error Response Infrastructure

Here's complete infrastructure for generating standardized JSON-RPC error responses:

```
// error-types.ts - Complete error type definitions
```

TYPESCRIPT

```
export enum JsonRpcErrorCode {  
  
    PARSE_ERROR = -32700,  
  
    INVALID_REQUEST = -32600,  
  
    METHOD_NOT_FOUND = -32601,  
  
    INVALID_PARAMS = -32602,  
  
    INTERNAL_ERROR = -32603  
  
}  
  
  
export interface JsonRpcError {  
  
    code: JsonRpcErrorCode;  
  
    message: string;  
  
    data?: any;  
  
}  
  
  
export interface ErrorContext {  
  
    requestId?: string | number;  
  
    method?: string;  
  
    originalError?: Error;  
  
    additionalInfo?: Record<string, any>;  
  
}  
  
  
export class LspError extends Error {  
  
    constructor(  
  
        public code: JsonRpcErrorCode,  
  
        public message: string,  
  
        public data?: any,  
  
        public context?: ErrorContext  
  
    ) {
```

```
super(message);

this.name = 'LspError';

}

}

// error-handler.ts - Complete error response generation

export class ErrorHandler {

private errorCounts = new Map<string, number>();

private lastErrors = new Map<string, Date>();

/** 

 * Converts exceptions into appropriate JSON-RPC error responses

 */

public handleError(error: Error, context: ErrorContext): JsonRpcResponse {

    // Track error frequency for degradation decisions

    this.trackError(error, context);

    const jsonRpcError = this.classifyError(error, context);

    return {

        jsonrpc: JSONRPC_VERSION,

        id: context.requestId || null,

        error: jsonRpcError

    };

}

/** 

 * Classifies errors into appropriate JSON-RPC error codes

 */

}
```

```
private classifyError(error: Error, context: ErrorContext): JsonRpcError {

    // TODO 1: Check if error is already a LspError with explicit code

    // TODO 2: Examine error message/type for parse error indicators

    // TODO 3: Check context.method against supported method list

    // TODO 4: Validate request structure for invalid request detection

    // TODO 5: Default to internal error for unclassified exceptions

    // TODO 6: Include appropriate error data based on error type

    // TODO 7: Sanitize error messages to avoid information leakage

}

/***
 * Tracks error patterns for capability degradation decisions
 */
private trackError(error: Error, context: ErrorContext): void {

    const errorKey = `${context.method || 'unknown'}:${error.constructor.name}`;

    const currentCount = this.errorCounts.get(errorKey) || 0;

    this.errorCounts.set(errorKey, currentCount + 1);

    this.lastErrors.set(errorKey, new Date());
}

/***
 * Determines if a capability should be degraded due to error frequency
 */
public shouldDegradeCapability(method: string): boolean {

    // TODO 1: Get error count for method over recent time window

    // TODO 2: Compare against degradation threshold

    // TODO 3: Consider error severity and impact

    // TODO 4: Return true if capability should be temporarily disabled
}
```

```
    }  
  
}
```

Transport Layer Error Detection

Here's complete infrastructure for detecting and handling transport-layer failures:

```
// connection-monitor.ts - Complete connection health monitoring
```

TYPESCRIPT

```
export enum ConnectionState {  
  
  DISCONNECTED = 'disconnected',  
  
  CONNECTING = 'connecting',  
  
  CONNECTED = 'connected',  
  
  DEGRADED = 'degraded',  
  
  FAILED = 'failed'  
  
}
```

```
export interface ConnectionHealth {  
  
  state: ConnectionState;  
  
  lastMessageTime: Date;  
  
  errorCount: number;  
  
  bytesSent: number;  
  
  bytesReceived: number;  
  
}
```

```
export class ConnectionMonitor {  
  
  private state = ConnectionState.DISCONNECTED;  
  
  private health: ConnectionHealth = {  
  
    state: ConnectionState.DISCONNECTED,  
  
    lastMessageTime: new Date(),  
  
    errorCount: 0,  
  
    bytesSent: 0,  
  
    bytesReceived: 0  
  
  };  
  
  private stateChangeListeners: Array<(state: ConnectionState) => void> = [];  
  
  /**
```

```
* Updates connection state based on transport events

*/
public updateState(newState: ConnectionState, reason?: string): void {
    if (this.canTransition(this.state, newState)) {

        const oldState = this.state;

        this.state = newState;

        this.health.state = newState;

        this.notifyStateChange(oldState, newState, reason);

    }
}

/**
 * Records successful message transmission
*/
public recordMessage(direction: 'sent' | 'received', byteCount: number): void {
    this.health.lastMessageTime = new Date();

    if (direction === 'sent') {

        this.health.bytesSent += byteCount;

    } else {

        this.health.bytesReceived += byteCount;

    }

    // Successful message may improve degraded connection

    if (this.state === ConnectionState.DEGRADED) {

        this.considerStateImprovement();

    }
}
```

```
/**  
 * Records transport error for state management  
 */  
  
public recordError(error: Error, context: string): void {  
    this.health.errorCount++;  
  
    // TODO 1: Classify error severity (parse vs transport vs encoding)  
    // TODO 2: Update state based on error type and current state  
    // TODO 3: Log error details for debugging  
    // TODO 4: Notify listeners of potential state change  
    // TODO 5: Consider immediate failure vs degraded operation  
}  
  
private canTransition(from: ConnectionState, to: ConnectionState): boolean {  
    // TODO: Implement state machine transition validation  
    // Valid transitions: disconnected->connecting, connecting->connected, etc.  
}  
  
private considerStateImprovement(): void {  
    // TODO: Check if recent success rate justifies upgrading from degraded to connected  
}  
  
private notifyStateChange(oldState: ConnectionState, newState: ConnectionState, reason?: string): void {  
    this.stateChangeListeners.forEach(listener => listener(newState));  
}  
}
```

Parse Error Recovery Framework

Here's a skeleton implementation for syntax error recovery:

```
// parser-recovery.ts - Parse error recovery strategies
```

TYPESCRIPT

```
export enum RecoveryStrategy {  
  
  PANIC_MODE = 'panic_mode',  
  
  ERROR_PRODUCTION = 'error_production',  
  
  PHRASE_LEVEL = 'phrase_level',  
  
  BALANCED_DELIMITERS = 'balanced_delimiters'  
  
}
```

```
export interface RecoveryPoint {  
  
  position: number;  
  
  strategy: RecoveryStrategy;  
  
  confidence: number;  
  
  context: string;  
  
}
```

```
export interface ParseError {  
  
  position: number;  
  
  expected: string[];  
  
  actual: string;  
  
  message: string;  
  
  recoveryHints: string[];  
  
}
```

```
export class ParseRecoveryEngine {  
  
  private recoveryStrategies = new Map<RecoveryStrategy, (error: ParseError) =>  
    RecoveryPoint[]>();
```

```
  constructor() {  
  
    this.setupRecoveryStrategies();  
  
  }
```

```

/**
 * Attempts to recover from parse error using multiple strategies
 */

public recoverFromError(error: ParseError, tokens: Token[], currentPos: number): RecoveryPoint | null {
    // TODO 1: Try each recovery strategy in priority order
    // TODO 2: Score each potential recovery point by confidence
    // TODO 3: Select highest-confidence recovery that makes structural sense
    // TODO 4: Validate recovery doesn't create worse problems downstream
    // TODO 5: Return recovery point or null if no viable recovery found
    // Hint: Higher confidence = more tokens match expected patterns after recovery
}

/**
 * Implements panic mode recovery - skip to synchronization point
 */

private panicModeRecovery(error: ParseError, tokens: Token[], pos: number): RecoveryPoint[] {
    // TODO 1: Define synchronization tokens (semicolon, closing brace, keywords)
    // TODO 2: Scan forward to find next synchronization point
    // TODO 3: Ensure synchronization point makes sense in current parsing context
    // TODO 4: Return recovery point with appropriate confidence score
    // Hint: Confidence decreases with distance to sync point
}

/**
 * Implements phrase-level recovery - insert missing tokens
*/

```

```
private phraseLevelRecovery(error: ParseError, tokens: Token[], pos: number): RecoveryPoint[] {  
  
    // TODO 1: Analyze expected tokens to identify likely missing elements  
  
    // TODO 2: Check if inserting expected token creates valid parse  
  
    // TODO 3: Validate insertion doesn't conflict with following tokens  
  
    // TODO 4: Return recovery point with high confidence for obvious fixes  
  
    // Hint: Missing semicolon or comma usually has high confidence  
  
}  
  
private setupRecoveryStrategies(): void {  
  
    this.recoveryStrategies.set(RecoveryStrategy.PANIC_MODE,  
    this.panicModeRecovery.bind(this));  
  
    this.recoveryStrategies.set(RecoveryStrategy.PHRASE_LEVEL,  
    this.phraseLevelRecovery.bind(this));  
  
    // Add other strategies...  
  
}  
  
}
```

Partial AST Construction

Here's infrastructure for building AST nodes that represent partially parsed code:

```
// partial-ast.ts - Handles incomplete AST construction

export interface ErrorRegion {

    start: number;

    end: number;

    errorType: string;

    recoveryAttempted: boolean;

    originalText: string;

}
```

```
export interface PartialASTNode extends ASTNode {

    isComplete: boolean;

    errorRegions: ErrorRegion[];

    confidence: number;

    partialInfo?: Record<string, any>;

}
```

```
export class PartialASTBuilder {

    /**
     * Creates AST node marked as incomplete with error information
     */

    public createPartialNode(

        kind: string,

        range: Range,

        errorInfo: ErrorRegion[]

    ): PartialASTNode {

        // TODO 1: Create base AST node with provided kind and range

        // TODO 2: Mark node as incomplete and add error regions

        // TODO 3: Calculate confidence score based on successfully parsed portions
    }
}
```

TYPESCRIPT

```

    // TODO 4: Extract any partial information that was successfully parsed

    // TODO 5: Set up parent-child relationships for completed portions

    // Hint: Confidence = (successfully parsed tokens) / (total expected tokens)

}

/***
 * Attempts to extract semantic information from incomplete nodes
 */

public extractPartialSemantics(node: PartialASTNode): Record<string, any> {

    // TODO 1: Identify which semantic elements are reliably parsed

    // TODO 2: Extract symbol names, partial type information, scope indicators

    // TODO 3: Mark extracted information with confidence levels

    // TODO 4: Return structured partial semantic data

    // Hint: Even incomplete function has name and parameter start

}

}

```

Milestone Checkpoints

After Milestone 1 (JSON-RPC & Initialization):

- Test malformed JSON messages: `echo '{"invalid": json}' | node dist/server.js`
- Expected: Server sends parse error response and continues processing
- Test missing Content-Length: Send raw JSON without headers
- Expected: Server detects framing error and requests retransmission

After Milestone 2 (Document Synchronization):

- Test invalid document URI: Send didOpen with malformed file URI
- Expected: Server responds with invalid params error, continues processing other documents
- Test version number regression: Send didChange with older version number
- Expected: Server detects version mismatch and requests document resync

After Milestone 3 (Language Features):

- Test completion on syntax error: Request completion inside malformed code
- Expected: Server returns partial completions with uncertainty indicators

- Test hover on incomplete declaration: Hover over partially parsed symbol
- Expected: Server returns available information marked as incomplete

After Milestone 4 (Diagnostics & Code Actions):

- Test diagnostic publishing with parse errors: Introduce syntax error in document
- Expected: Server publishes syntax diagnostic and continues providing features for valid code
- Test code action on unsupported error: Request quick fix for complex semantic error
- Expected: Server responds with no applicable actions rather than crashing

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Server stops responding	Unhandled exception in message handler	Check stderr for stack traces	Add try-catch in all handlers
Client shows "server crashed"	Process exit due to protocol error	Check exit code and final log messages	Implement graceful error recovery
Features work intermittently	Parse errors affecting symbol resolution	Check diagnostic output for parse failures	Improve error recovery strategies
High memory usage	Error recovery creating too many partial AST nodes	Profile memory usage during error scenarios	Limit partial node creation depth
Slow response times	Error handling triggering expensive recovery operations	Profile request handling with errors present	Cache recovery results, limit retries

Testing Strategy

Milestone(s): This section spans all milestones with specific testing approaches for Milestone 1 (JSON-RPC & Initialization) protocol compliance, Milestone 2 (Document Synchronization) state management, Milestone 3 (Language Features) functionality verification, and Milestone 4 (Diagnostics & Code Actions) integration testing.

Think of testing an LSP server like stress-testing a restaurant kitchen during rush hour. You need to verify that individual stations work correctly (unit tests), that orders flow smoothly between stations (integration tests), that the kitchen can handle peak load without breaking down (performance tests), and that customers receive exactly what they ordered (protocol compliance tests). The key insight is that LSP servers are **stateful, concurrent systems** that must maintain document consistency while serving multiple overlapping requests - this creates testing challenges that don't exist in simple request-response services.

Our testing strategy addresses three critical dimensions: **protocol compliance** (does our server speak LSP correctly?), **language feature accuracy** (do completions, hover, and diagnostics return correct results?), and

performance characteristics (can we handle large files and many concurrent requests without degrading?).

Each dimension requires different testing approaches because they validate different failure modes.

Milestone Checkpoints

Each milestone represents a significant capability increment that can be independently verified. The checkpoint approach ensures we catch integration issues early rather than discovering them during final testing when they're expensive to fix.

Milestone 1: JSON-RPC & Initialization Checkpoint

The first milestone validates that our server can establish communication with language clients and negotiate capabilities correctly. Think of this as proving our restaurant can take orders and tell customers what's on the menu before we worry about cooking anything.

The core verification points focus on **message framing correctness** and **protocol state management**. Many LSP implementations fail here because Content-Length calculation seems trivial but has subtle edge cases around UTF-8 encoding and CRLF handling.

Test Category	Verification Method	Expected Behavior	Failure Indicators
Message Framing	Send malformed messages, partial messages, multiple concatenated messages	Server extracts complete messages, ignores malformed ones, doesn't crash	Server hangs, crashes, or responds to partial messages
Initialization Handshake	Send initialize request with various client capabilities	Server responds with appropriate ServerCapabilities, transitions to initialized state	Server claims unsupported capabilities, accepts requests before initialization
Capability Negotiation	Test with minimal client capabilities and maximal capabilities	Server only advertises capabilities it can fulfill	Server crashes on unsupported capability requests, advertises false capabilities
Shutdown Sequence	Send shutdown request followed by exit notification	Server responds to shutdown, stops processing requests, terminates cleanly on exit	Server continues processing after shutdown, hangs on exit, crashes

The **protocol state validation** is particularly critical. Create a test that sends requests out of order (completion before initialize, requests after shutdown) and verify the server responds with appropriate JSON-RPC errors rather than crashing or processing invalid requests.

```
// Example test sequence for state validation
```

TYPESCRIPT

1. Send completion `request` (should get `METHOD_NOT_FOUND` or server not initialized error)
2. Send initialize `request` (should succeed)
3. Send completion `request` (should get `METHOD_NOT_FOUND` - not initialized yet)
4. Send initialized notification
5. Send completion `request` (should succeed or return empty results)
6. Send shutdown `request` (should succeed)
7. Send completion `request` (should fail - server shutting down)
8. Send exit `notification` (server should terminate)

Milestone 2: Document Synchronization Checkpoint

Document synchronization is where many LSP servers develop subtle bugs that only surface under concurrent load. Think of this like verifying that our restaurant's order tickets stay accurate even when customers change their minds mid-order.

The critical validation focuses on **incremental change application** and **version consistency**. Document synchronization bugs often manifest as "works fine with small changes but corrupts documents with large edits" or "works in single-threaded testing but fails with concurrent document updates."

Test Scenario	Test Method	Expected Outcome	Common Bug Indicators
Full Document Sync	Open document, verify stored content matches exactly	Document text identical to client, correct URI and version tracking	Text corruption, encoding issues, version mismatches
Incremental Range Edits	Apply series of range-based edits, verify final document state	Each edit applied correctly, final text matches expected result	Off-by-one errors, incorrect range calculations, cumulative drift
Multi-document Management	Open multiple documents, edit concurrently, close selectively	Each document maintained independently, no cross-document contamination	Document confusion, memory leaks, stale document references
Version Race Conditions	Send rapid document changes with version numbers	Server rejects out-of-order versions, maintains consistency	Accepts stale updates, version numbers become inconsistent

The **incremental synchronization test** should include challenging scenarios like inserting text at document beginning (shifts all subsequent positions), replacing large ranges with smaller text (document shrinks), and Unicode characters that occupy multiple UTF-16 code units.

Create a test document with known line/character positions and apply a series of edits while verifying that `positionToOffset` and `offsetToPosition` conversions remain accurate throughout the change sequence.

Milestone 3: Language Features Checkpoint

Language features represent the core value proposition of the LSP server - this is where we prove our restaurant can actually cook the food, not just take orders. The testing challenge is that language features depend on **semantic understanding** that can fail in subtle ways.

Think of testing language features like testing a GPS navigation system - you need to verify it works on highways (simple cases), city streets (moderate complexity), and construction zones (edge cases with incomplete information).

Feature	Test Approach	Success Criteria	Regression Indicators
Completion	Position cursor at various code locations, request completions	Returns contextually appropriate suggestions, excludes out-of-scope symbols	Returns irrelevant suggestions, missing obvious candidates, includes inaccessible symbols
Hover	Hover over symbols, keywords, operators	Returns type information and documentation when available	Empty responses for valid symbols, crashes on invalid positions, incorrect type information
Go-to-Definition	Click on symbol references throughout codebase	Navigates to correct declaration location	Wrong locations, crashes on built-in symbols, fails across file boundaries
Find References	Request references for various symbol types	Returns all usage locations, excludes false positives	Misses references, includes unrelated symbols, performance degradation on large files

The **completion testing** requires careful attention to **context sensitivity**. A robust test suite includes:

1. **Scope-based filtering:** Variables declared in inner scopes should appear in completions, but variables from sibling scopes should not
2. **Type-based filtering:** Method completions should only include methods available on the target type
3. **Keyword completion:** Language keywords should appear in appropriate syntactic positions
4. **Import-based completion:** Symbols from imported modules should be available with correct qualification

Create test files with known symbol relationships and verify that completion results match the expected symbol visibility rules for your target language.

Milestone 4: Diagnostics & Code Actions Checkpoint

Diagnostics and code actions represent the **proactive assistance** capabilities of the LSP server - like a restaurant kitchen that warns you about food allergies and suggests menu modifications. This milestone tests both **error detection accuracy** and **fix suggestion quality**.

The challenge is that diagnostic systems can produce false positives (crying wolf), false negatives (missing real problems), or performance issues (taking too long to analyze).

Test Category	Verification Method	Quality Indicators	Problem Signs
Syntax Error Detection	Introduce known syntax errors	Errors reported at correct locations with helpful messages	Wrong error positions, confusing error messages, crashes on malformed code
Semantic Error Detection	Create type mismatches, undefined references	Semantic errors caught with appropriate severity levels	False positives on valid code, missing obvious type errors
Diagnostic Performance	Open large files, make changes	Diagnostics update within reasonable time, don't block editor	Long delays, memory spikes, editor freezing
Code Action Relevance	Request code actions for various diagnostic types	Suggested fixes are applicable and correct	Inapplicable fixes, fixes that break code, missing obvious fixes

The **diagnostic accuracy test** should include edge cases like:

1. **Parse error recovery**: Syntax errors should not prevent analysis of unaffected code sections
2. **Incremental re-analysis**: Changes should only trigger re-analysis of affected scopes, not entire files
3. **Cross-file dependencies**: Changes in one file should trigger diagnostic updates in dependent files
4. **Diagnostic cleanup**: Closing documents should remove their diagnostics from the client

Create a test that introduces an error, verifies the diagnostic appears, fixes the error, and confirms the diagnostic disappears within a reasonable time window.

Integration Testing

Integration testing validates that our LSP server works correctly with **real language clients** in realistic usage scenarios. Think of this as moving from controlled kitchen testing to serving actual customers in a busy restaurant environment.

The fundamental challenge is that LSP clients (VS Code, Neovim, Emacs, etc.) each have slightly different interpretation quirks and performance characteristics. Integration testing ensures our server doesn't just work with our test harness, but with the actual tools developers use daily.

Client Compatibility Testing

Different language clients stress the LSP server in different ways. VS Code tends to send rapid bursts of requests during typing, Neovim may send fewer but more complex requests, and command-line tools might open many files simultaneously.

Client Type	Testing Focus	Key Scenarios	Success Metrics
VS Code	Real-time responsiveness, incremental sync	Rapid typing, large file editing, multi-file projects	No typing lag, accurate diagnostics, completion works during heavy editing
Neovim/Vim	Batch operations, stability	Opening many files, bulk edits, long-running sessions	No memory leaks, handles large operations, stable over hours of use
Command Line Tools	Bulk analysis, resource usage	Analyze entire project, generate reports, CI/CD integration	Completes successfully, reasonable resource usage, consistent results
Web-based IDEs	Network considerations, serialization	Remote file access, large message payloads	Handles network delays, efficient serialization, works with proxying

The **VS Code integration test** should simulate realistic editing patterns:

1. **Rapid typing simulation:** Send didChange notifications at 60+ WPM typing speed while requesting completions
2. **Multi-cursor editing:** Apply changes to multiple document locations simultaneously
3. **Large file handling:** Open files with 10,000+ lines and verify responsiveness doesn't degrade
4. **Project-wide operations:** Trigger find-all-references across a project with 100+ files

Performance Integration Testing

Real-world LSP servers must handle **concurrent requests** while maintaining **low latency** for interactive features. Think of this like stress-testing a restaurant during lunch rush - you need to serve many customers quickly without mixing up orders.

Performance integration testing reveals bottlenecks that don't appear in unit tests because it exercises the full system under realistic load patterns.

Performance Dimension	Test Approach	Acceptable Thresholds	Failure Indicators
Completion Latency	Measure time from completion request to response	< 100ms for most cases, < 500ms for complex analysis	Completions take seconds, editor becomes unresponsive
Diagnostic Timeliness	Time from document change to diagnostic publication	< 2 seconds for syntax, < 10 seconds for semantics	Diagnostics appear minutes later, memory usage grows unbounded
Memory Usage	Monitor memory consumption during extended sessions	Stable memory usage, < 500MB for typical projects	Memory leaks, unbounded cache growth, crashes due to OOM
Concurrent Request Handling	Send multiple overlapping requests	All requests receive responses, no request starvation	Some requests never respond, responses mixed up between requests

The **load testing approach** should include:

1. **Gradual ramp-up:** Start with single requests, gradually increase concurrency to find breaking points
2. **Realistic request patterns:** Mix quick requests (completion) with expensive ones (find-all-references)
3. **Long-running stability:** Run tests for hours to detect memory leaks and resource exhaustion
4. **Recovery testing:** Introduce failures and verify the server recovers gracefully

Protocol Compliance Verification

LSP protocol compliance ensures our server works with current and future language clients. Think of this as ensuring our restaurant follows health codes - it's not just about today's customers, but about maintaining standards that work everywhere.

Many subtle protocol violations only surface when testing with different client implementations or newer protocol versions.

Compliance Area	Testing Method	Verification Points	Non-compliance Symptoms
Message Format	Send edge-case messages, validate responses	All responses are valid JSON-RPC, correct LSP structure	Malformed JSON, missing required fields, incorrect types
Error Handling	Trigger various error conditions	Errors use correct JSON-RPC error codes and have helpful messages	Generic errors, wrong error codes, server crashes instead of error responses
Capability Contracts	Test features based on advertised capabilities	Server only provides features it advertised, handles unsupported requests gracefully	Claims support but fails, processes requests for unadvertised features
Protocol Versioning	Test with different LSP protocol versions	Graceful degradation with older clients, forward compatibility	Breaks with older clients, crashes on unknown fields

End-to-End Workflow Testing

Real developers use LSP servers in complex workflows that combine multiple features. End-to-end testing validates these **interaction patterns** work correctly together.

Workflow	Test Scenario	Expected Flow	Integration Issues
Code Refactoring	Rename symbol, verify references update	Find references → rename → update all locations → diagnostics clear	Stale references, broken diagnostics after rename
Error Fix Cycle	Introduce error → get diagnostic → apply code action → verify fix	Error appears → code action suggested → action applied → error disappears	Code actions don't fix the reported problem
Multi-file Navigation	Go to definition across files → make changes → return	Definition found in other file → changes applied → original file reflects updates	Cross-file changes not reflected, navigation breaks
Project Analysis	Open project → wait for analysis → explore codebase	All files analyzed → diagnostics appear → completion works everywhere	Incomplete analysis, missing cross-file information

The **refactoring workflow test** is particularly revealing because it exercises symbol resolution, cross-file dependencies, and state consistency all together:

```
// Example refactoring test flow
```

TYPESCRIPT

1. Open main file containing **function call**
2. Go to definition (**should jump to function definition in another file**)
3. Rename function at definition site
4. Verify all call sites show updated name in completions
5. Verify no "undefined function" diagnostics appear
6. Apply additional edits and confirm consistency maintained

Implementation Guidance

The testing strategy requires both **automated test infrastructure** and **manual verification procedures**. Think of automated tests as your restaurant's food safety sensors - they catch problems quickly and consistently - while manual testing is like having an experienced chef taste-test dishes to ensure quality.

Technology Recommendations

Testing Layer	Simple Option	Advanced Option
Unit Testing	Jest/Vitest with TypeScript	Jest + custom LSP test utilities
Integration Testing	Manual client testing + scripts	Automated client simulation framework
Protocol Testing	JSON schema validation + manual	LSP protocol compliance test suite
Performance Testing	Simple timing measurements	Continuous performance benchmarking
Load Testing	Sequential request scripts	Concurrent request simulation tools

Test Infrastructure Setup

The testing infrastructure should mirror the development structure while providing utilities for LSP-specific testing challenges like message framing and protocol state management.

```
project-root/
  src/
    transport/
    protocol/
    document-manager/
    language-engine/
    feature-providers/
  test/
    unit/           ← isolated component tests
      transport.test.ts
      document-manager.test.ts
      language-engine.test.ts
    integration/   ← cross-component tests
      initialization.test.ts
      document-sync.test.ts
      language-features.test.ts
    protocol/      ← LSP compliance tests
      protocol-compliance.test.ts
      client-compatibility/
    fixtures/       ← test documents and data
      sample-code/
      protocol-messages/
    utilities/     ← testing helper utilities
      lsp-test-client.ts
      message-assertions.ts
      performance Helpers.ts
```

LSP Test Client Infrastructure

Most LSP testing requires simulating a language client that can send properly formatted JSON-RPC messages and validate responses. This infrastructure is complex enough that it should be built once and reused across all tests.

```
// Complete LSP test client for protocol testing

import { spawn, ChildProcess } from 'child_process';

import { MessageFramer } from '../src/transport/message-framer';

export class LspTestClient {

    private process: ChildProcess;

    private framer: MessageFramer;

    private responseHandlers: Map<string | number, (response: any) => void>;

    private notificationHandlers: Map<string, (params: any) => void>;

    private requestId: number = 1;

    constructor(serverCommand: string, serverArgs: string[]) {

        this.process = spawn(serverCommand, serverArgs, {

            stdio: ['pipe', 'pipe', 'inherit']

        });

        this.framer = new MessageFramer();

        this.responseHandlers = new Map();

        this.notificationHandlers = new Map();

        this.setupMessageHandling();

    }

    private setupMessageHandling(): void {

        this.process.stdout!.on('data', (chunk: Buffer) => {

            this.framer.handleData(chunk.toString('utf8'));




            let message;

            while ((message = this.framer.tryExtractMessage()) !== null) {

                this.handleMessage(JSON.parse(message));

            }

        });

    }

}
```

TYPESCRIPT

```
});

}

private handleMessage(message: any): void {

    if (message.id !== undefined && (message.result !== undefined || message.error !== undefined)) {

        // This is a response

        const handler = this.responseHandlers.get(message.id);

        if (handler) {

            handler(message);

            this.responseHandlers.delete(message.id);

        }

    } else if (message.method !== undefined) {

        // This is a notification or request from server

        const handler = this.notificationHandlers.get(message.method);

        if (handler) {

            handler(message.params);

        }

    }

}

async sendRequest(method: string, params: any): Promise<any> {

    const id = this.requestId++;

    const message = {

        jsonrpc: '2.0',

        id: id,

        method: method,

        params: params

    };

}
```

```
        return new Promise((resolve, reject) => {

            this.responseHandlers.set(id, (response) => {

                if (response.error) {

                    reject(new Error(`LSP Error: ${response.error.message}`));

                } else {

                    resolve(response.result);

                }

            });

            const frameMessage = this.frameMessage(JSON.stringify(message));

            this.process.stdin!.write(frameMessage);

        });

    }

    sendNotification(method: string, params: any): void {

        const message = {

            jsonrpc: '2.0',

            method: method,

            params: params

        };

        const framedMessage = this.frameMessage(JSON.stringify(message));

        this.process.stdin!.write(framedMessage);

    }

    onNotification(method: string, handler: (params: any) => void): void {

        this.notificationHandlers.set(method, handler);

    }

}
```

```

private frameMessage(content: string): string {
    const contentBytes = Buffer.byteLength(content, 'utf8');

    return `Content-Length: ${contentBytes}\r\n\r\n${content}`;
}

async initialize(clientCapabilities: any = {}): Promise<any> {
    const result = await this.sendRequest('initialize', {
        processId: process.pid,
        clientInfo: { name: 'LSP Test Client', version: '1.0.0' },
        capabilities: clientCapabilities,
        workspaceFolders: null
    });

    this.sendNotification('initialized', {});

    return result;
}

async shutdown(): Promise<void> {
    await this.sendRequest('shutdown', null);

    this.sendNotification('exit', null);

    return new Promise((resolve) => {
        this.process.on('exit', () => resolve());
    });
}

```

Milestone Testing Procedures

Each milestone should have a **specific testing checklist** that can be executed to verify completion. These procedures bridge the gap between automated tests and manual verification.

```
// Milestone 1 Testing Skeleton
```

TYPESCRIPT

```
export class Milestone1Tests {

    private client: LspTestClient;

    async testInitializationHandshake(): Promise<void> {
        // TODO 1: Create test client connected to LSP server
        // TODO 2: Send initialize request with basic client capabilities
        // TODO 3: Verify server responds with ServerCapabilities
        // TODO 4: Send initialized notification
        // TODO 5: Verify server is ready to accept requests
        // TODO 6: Attempt to send completion request (should succeed or return empty)
        // Hint: Server should not crash or hang during this sequence
    }

    async testMessageFraming(): Promise<void> {
        // TODO 1: Send message with exactly correct Content-Length
        // TODO 2: Send message with incorrect Content-Length (should be ignored)
        // TODO 3: Send multiple messages concatenated together
        // TODO 4: Send message split across multiple data chunks
        // TODO 5: Verify server extracts and processes only valid messages
        // Hint: Use different message types to verify each was processed
    }

    async testShutdownSequence(): Promise<void> {
        // TODO 1: Initialize server normally
        // TODO 2: Send some requests to verify server is working
        // TODO 3: Send shutdown request, verify success response
        // TODO 4: Send request after shutdown (should fail)
        // TODO 5: Send exit notification
    }
}
```

```
// TODO 6: Verify server process terminates within 5 seconds  
  
// Hint: Monitor server process exit code - should be 0  
  
}  
  
}
```

Performance Testing Utilities

Performance testing for LSP servers requires measuring **latency distributions** and **resource usage patterns** under realistic load. Simple average measurements miss the tail latency problems that make editors feel sluggish.

```
// Performance measurement utilities for LSP testing
```

TYPESCRIPT

```
export class PerformanceProfiler {

    private measurements: Map<string, number[]> = new Map();

    private startTimes: Map<string, number> = new Map();

    startMeasurement(operation: string): void {
        this.startTimes.set(operation, performance.now());
    }

    endMeasurement(operation: string): void {
        const startTime = this.startTimes.get(operation);
        if (startTime) {
            const duration = performance.now() - startTime;
            if (!this.measurements.has(operation)) {
                this.measurements.set(operation, []);
            }
            this.measurements.get(operation)!.push(duration);
            this.startTimes.delete(operation);
        }
    }

    getPerformanceReport(operation: string): PerformanceReport {
        // TODO 1: Calculate min, max, mean, median, p95, p99 latencies
        // TODO 2: Identify any measurements above acceptable thresholds
        // TODO 3: Return structured report with recommendations
        // Hint: Use percentile calculations to identify tail latency issues
        throw new Error("Implementation needed");
    }
}
```

```

async runConcurrencyTest(
    testFunction: () => Promise<void>,
    concurrency: number,
    duration: number
): Promise<ConcurrencyTestResult> {
    // TODO 1: Start multiple concurrent test functions
    // TODO 2: Track completion times and success/failure rates
    // TODO 3: Monitor for request mixing or starvation
    // TODO 4: Return detailed concurrency performance report
    // Hint: Use Promise.allSettled to handle individual failures
    throw new Error("Implementation needed");
}

}

interface PerformanceReport {
    operation: string;
    sampleCount: number;
    meanLatency: number;
    medianLatency: number;
    p95Latency: number;
    p99Latency: number;
    maxLatency: number;
    acceptableLatency: boolean;
    recommendations: string[];
}

```

Debugging Integration Test Failures

Integration test failures in LSP servers often manifest as **timing-dependent issues** or **state inconsistencies** that are difficult to reproduce. Having systematic debugging procedures saves hours of investigation.

Symptom	Likely Cause	Diagnostic Steps	Resolution
Tests pass individually but fail in suite	Shared state between tests	Add cleanup verification, check for leaked event listeners	Reset server state between tests, ensure proper cleanup
Intermittent timeout failures	Race conditions in async operations	Add detailed timing logs, test with different delays	Add proper synchronization, use event-driven waiting instead of fixed delays
Different behavior in CI vs local	Environment differences	Compare Node.js versions, check stdio buffering differences	Standardize test environment, add environment validation
Protocol violations only with certain clients	Client-specific interpretation differences	Test same scenarios with multiple LSP clients, compare message logs	Add client compatibility layer, handle edge cases in protocol interpretation

Debugging Guide

Milestone(s): This section spans all milestones with particular emphasis on Milestone 1 (JSON-RPC & Initialization) for protocol debugging, Milestone 2 (Document Synchronization) for state consistency issues, Milestone 3 (Language Features) for symbol resolution problems, and Milestone 4 (Diagnostics & Code Actions) for performance optimization.

Think of debugging an LSP server like troubleshooting a conversation between two people who speak different dialects of the same language through an unreliable phone connection. The language client (editor) and language server are trying to coordinate complex operations, but the communication channel (JSON-RPC over stdin/stdout) introduces timing issues, encoding problems, and message boundaries that don't align with how data arrives. When something goes wrong, you need to determine whether the problem is in the "phone connection" (transport layer), the "translation" (protocol handling), the "memory" (document synchronization), or the "reasoning" (language analysis).

Most LSP debugging falls into two major categories: **protocol debugging** (communication and state management issues) and **performance debugging** (responsiveness and scalability problems). Protocol issues typically manifest as initialization failures, message parsing errors, or inconsistent document state. Performance issues show up as editor lag, high CPU usage, memory leaks, or timeouts during complex operations.

The debugging process requires understanding the **bidirectional communication** patterns between client and server. Unlike traditional request-response systems, LSP involves both synchronous requests (like completion or hover) and asynchronous notifications (like document changes or diagnostic publishing). This creates complex interaction patterns where problems in one area can cascade into seemingly unrelated failures.

Protocol Debugging

Protocol debugging focuses on diagnosing JSON-RPC communication issues, initialization problems, and protocol state management failures. These issues typically occur at the boundary between the language client and server, involving message framing, capability negotiation, or lifecycle management.

Message Framing Issues

The most common protocol issue involves **Content-Length framing** problems in the `StreamTransport` and `MessageFramer` components. The JSON-RPC over stdio transport uses HTTP-style headers to delimit message boundaries, but this creates several failure modes that are difficult to diagnose without proper tooling.

Think of message framing like parsing sentences from a continuous stream of letters without spaces. The `CONTENT_LENGTH_HEADER` acts like punctuation marks, telling you where one message ends and the next begins. When framing fails, you either miss message boundaries (combining two messages into one) or split messages incorrectly (treating part of a message as a complete message).

Symptom	Root Cause	Diagnostic Approach	Resolution
Server hangs on startup	Malformed initialize request	Log raw stdin bytes, check Content-Length calculation	Verify client sends CRLF line endings, not just LF
Random JSON parse errors	Message boundaries crossing buffer reads	Add message boundary logging to <code>handleData</code>	Implement complete message buffering before parsing
Intermittent request failures	Character encoding mismatch	Compare byte counts with character counts	Ensure UTF-8 encoding consistency throughout pipeline
Server exits unexpectedly	Uncaught exception in message parsing	Add try-catch around <code>processMessage</code> calls	Implement graceful error recovery with JSON-RPC error responses

The `MessageFramer` maintains a `buffer` and `contentLength` state to accumulate partial messages across multiple `handleData` calls. A common pitfall occurs when developers assume that each stdin read contains exactly one complete message. In reality, the operating system may deliver data in arbitrary chunks - sometimes multiple messages in one read, sometimes a single message split across several reads.

⚠ Pitfall: Content-Length Calculation Errors Many developers calculate Content-Length using string length instead of byte length. In UTF-8, multi-byte characters cause the byte count to exceed the character count, leading to message truncation. Always use `Buffer.byteLength(jsonString, 'utf8')` rather than `jsonString.length` when calculating Content-Length headers.

The debugging approach for framing issues involves adding detailed logging to the `tryExtractMessage` method to track buffer state, content length parsing, and message extraction boundaries. Create a diagnostic mode that logs every byte received, every header parsed, and every message boundary detected.

Initialization and Capability Negotiation Problems

The LSP **initialization handshake** involves a precisely choreographed sequence of messages that establish server capabilities and protocol state. The `ProtocolState` component manages transitions from `Uninitialized` through `Initializing`, `Initialized`, and eventually `Shutdown` states. Failures in this sequence often stem from incorrect capability negotiation or premature message sending.

The mental model for initialization debugging is like a formal diplomatic protocol where each party must follow specific steps in exact order. The client sends an `initialize` request with its `ClientCapabilities`, the server responds with `ServerCapabilities`, and only after the client sends an `initialized` notification can normal LSP operations begin.

Initialization State	Expected Event	Common Failure Modes	Diagnostic Questions
Uninitialized	initialize request	Client sends other requests first	Is server rejecting pre-initialization requests with correct error code?
Initializing	Server processes capabilities	Server capabilities don't match implementation	Does server advertise only features it actually implements?
Initialized	initialized notification received	Client sends requests before initialized	Is server buffering or rejecting premature requests?
Ready	Normal LSP operations	Feature requests fail unexpectedly	Do server capabilities match what client is requesting?

A frequent issue occurs when the `CapabilityNegotiator` advertises features in `ServerCapabilities` that the server doesn't actually implement, or when the server attempts to use client features without checking if the client declared support for them. This creates a mismatch between negotiated capabilities and actual behavior.

Decision: Fail-Fast Capability Validation

- **Context:** Servers often crash or behave unpredictably when clients request unadvertised features or servers use undeclared client capabilities
- **Options:** 1) Graceful degradation with warnings, 2) Fail-fast validation that rejects unsupported requests, 3) Best-effort implementation that ignores capability mismatches
- **Decision:** Implement fail-fast validation during development with optional graceful degradation for production
- **Rationale:** Early detection of capability mismatches prevents harder-to-debug runtime failures and ensures protocol compliance
- **Consequences:** More predictable behavior but requires careful capability management and comprehensive error handling

The debugging approach for initialization problems involves creating a state transition log that tracks every protocol state change, the triggering event, and any capability mismatches. Implement validation that checks whether incoming requests are appropriate for the current `ServerState` and whether they require capabilities that weren't negotiated.

Document Synchronization Failures

Document synchronization issues in the `DocumentManager` component create some of the most frustrating debugging scenarios because they involve race conditions, version mismatches, and coordinate system problems that only manifest under specific timing conditions.

Think of document synchronization like maintaining synchronized copies of a rapidly changing document between two people editing collaboratively. The `TextDocumentItem` represents the server's understanding of the document, while the editor maintains its own copy. The `version` numbers act like revision timestamps to detect when the copies have diverged.

The most common synchronization problems involve **incremental synchronization** where the server applies `TextDocumentContentChangeEvent` objects incorrectly, leading to document content that differs from the editor's version. This creates a cascade of failures where symbol resolution, diagnostics, and language features all operate on incorrect document content.

Synchronization Issue	Detection Method	Impact	Resolution Strategy
Version number mismatch	Compare expected vs received version in <code>applyChanges</code>	All features work on stale content	Implement version validation with client resync request
Position coordinate errors	Validate ranges are within document bounds	Range-based operations fail or corrupt	Add boundary checking to <code>positionToOffset</code> conversions
Incremental edit application failures	Hash document content after changes	Gradual content divergence	Log detailed change application steps for debugging
Unicode handling problems	Compare byte offsets with character positions	Features work on wrong text locations	Ensure consistent UTF-16 code unit handling

The `DocumentSnapshot` mechanism helps isolate document versions for analysis, but introduces its own complexity around cache invalidation and memory management. A common debugging technique involves creating a document content verification system that periodically requests the full document content from the client and compares it with the server's cached version.

⚠ Pitfall: Position Coordinate Confusion LSP uses UTF-16 code units for positions, but many languages use byte offsets or UTF-8 character positions internally. The `offsetToPosition` and `positionToOffset`

conversion functions must account for multi-byte characters and surrogate pairs. Emoji and non-ASCII characters frequently expose coordinate conversion bugs.

Error Response and Recovery Issues

The JSON-RPC error handling system requires careful management of error codes, error context, and graceful degradation. The `JsonRpcError` and `ErrorHandler` types provide structured error reporting, but many servers implement error handling incorrectly, leading to client confusion and protocol violations.

Think of error handling like being a customer service representative who needs to give helpful responses even when internal systems fail. The client doesn't care about internal server problems - it needs to know whether to retry the request, try a different approach, or inform the user that a feature is temporarily unavailable.

Error Category	JSON-RPC Code	When to Use	Client Expectation
Parse errors	<code>PARSE_ERROR</code> (-32700)	Malformed JSON received	Client has internal bug, should not retry
Invalid requests	<code>INVALID_REQUEST</code> (-32600)	Request structure invalid	Client has protocol violation, should not retry
Method not found	<code>METHOD_NOT_FOUND</code> (-32601)	Unsupported LSP method	Client should check capabilities, may retry after capability refresh
Invalid parameters	<code>INVALID_PARAMS</code> (-32602)	Correct method, wrong parameters	Client should validate parameters, may retry with corrections
Internal errors	<code>INTERNAL_ERROR</code> (-32603)	Server implementation bug	Client may retry with backoff or disable feature

The `handleError` method in the error handling system must classify errors appropriately and provide actionable error messages. A common mistake involves returning generic `INTERNAL_ERROR` codes for all failures, which prevents clients from implementing intelligent retry and fallback strategies.

The debugging approach for error handling involves implementing comprehensive error logging that captures the complete context of each failure, including the original request, server state, and error propagation path. Create error injection tests that deliberately trigger different error conditions to verify that the server responds with appropriate JSON-RPC error codes and messages.

Performance Debugging

Performance debugging focuses on identifying and resolving responsiveness issues that impact the user experience in the editor. LSP servers must maintain low latency for interactive features like completion and hover while efficiently processing large codebases and complex analysis tasks.

Latency and Responsiveness Issues

The primary performance challenge in LSP servers involves balancing thoroughness of analysis with responsiveness to user interactions. The `LanguageEngine` must perform parsing, symbol resolution, and semantic analysis quickly enough that features like completion feel instantaneous (typically under 100ms) while also providing comprehensive diagnostics and cross-file analysis.

Think of LSP performance debugging like optimizing a restaurant kitchen that must serve both fast orders (like drinks and appetizers) and complex meals (like multi-course dinners). The completion and hover features are like drink orders - customers expect them immediately. Diagnostics and cross-file analysis are like complex meals - customers will wait longer but expect higher quality results.

The key insight is that different LSP features have different **latency budgets** and **accuracy requirements**. Auto-completion must respond within 100-200ms to feel responsive, but can provide approximate results that are refined over time. Diagnostics can take several seconds to complete but must be comprehensive and accurate.

Feature	Target Latency	Acceptable Latency	Accuracy Requirements	Performance Strategy
Completion	<100ms	<300ms	High relevance, approximate types OK	Incremental symbol resolution, cached results
Hover	<200ms	<500ms	Complete type info, accurate documentation	Symbol table lookup, minimal analysis
Diagnostics	<1000ms	<5000ms	Complete error detection, no false positives	Full analysis, can be async/batched
Go-to-definition	<500ms	<1000ms	Exact location, no false results	Symbol resolution with cross-file lookup
Find references	<2000ms	<10000ms	Complete result set, no missed references	Comprehensive workspace scan, can show progress

The `PerformanceProfiler` component tracks latency statistics for each operation type, measuring both individual request latency and overall system responsiveness. The debugging approach involves identifying which operations exceed their latency budgets and whether the problem is algorithmic complexity, inefficient data structures, or resource contention.

A common performance anti-pattern involves performing full document re-parsing and analysis for every request, rather than implementing **incremental analysis** strategies. The `Language Engine` should cache parsed ASTs, symbol tables, and analysis results, invalidating only the portions affected by document changes.

⚠ Pitfall: Blocking the Event Loop Many LSP server implementations perform synchronous analysis directly in request handlers, blocking all other operations until analysis completes. This creates terrible user experience where typing in the editor becomes sluggish during complex analysis operations. Implement asynchronous analysis with proper concurrency management.

Memory Usage and Resource Management

LSP servers often exhibit memory leak patterns because they maintain long-running processes that accumulate document content, cached analysis results, and symbol tables over extended editing sessions. The `DocumentManager` and `LanguageEngine` components must implement intelligent cache management and resource cleanup strategies.

The mental model for LSP memory debugging is like managing a research library that must keep frequently accessed books readily available while periodically returning older books to storage to make room for new acquisitions. The **cache invalidation** strategy must balance memory usage with analysis performance.

Memory Consumer	Growth Pattern	Cleanup Strategy	Monitoring Approach
Document content	Linear with open files	Remove on didClose, periodic cleanup of stale documents	Track document count and total content size
Parsed ASTs	Exponential with document size and complexity	LRU eviction, invalidate on document changes	Monitor AST cache hit rates and memory per AST
Symbol tables	Linear with codebase size	Incremental updates, remove unreferenced symbols	Track symbol count and cross-reference table size
Diagnostic results	Linear with error count	Replace on re-analysis, clear for closed documents	Monitor diagnostic count and payload size
Completion caches	Linear with API surface	Time-based expiration, invalidate on relevant changes	Track cache size and hit rates per document

The `CacheEntry` type supports multi-tier caching with `hotCapacity` for frequently accessed items and `warmCapacity` for less frequently used data. The cache management strategy should promote frequently accessed items to higher tiers and demote stale items to lower tiers before eviction.

Memory profiling for LSP servers requires understanding the **document lifecycle** and **analysis caching patterns**. Implement memory usage tracking that correlates memory growth with specific operations, document sizes, and cache hit rates. Create memory pressure tests that simulate extended editing sessions with large codebases.

Decision: Adaptive Cache Management

- **Context:** Static cache sizes work poorly for LSP servers handling diverse project sizes and usage patterns
- **Options:** 1) Fixed cache limits based on configuration, 2) Adaptive limits based on available memory, 3) No caching to avoid complexity
- **Decision:** Implement adaptive cache management that adjusts limits based on memory pressure and usage patterns
- **Rationale:** LSP servers must handle projects ranging from small scripts to large enterprise codebases with widely varying memory constraints
- **Consequences:** More complex cache implementation but better resource utilization and user experience across different environments

Concurrency and Threading Issues

LSP servers must handle concurrent requests while maintaining document state consistency and avoiding race conditions in the analysis pipeline. The challenge involves coordinating between synchronous request processing, asynchronous document analysis, and periodic background tasks like diagnostic publishing.

Think of LSP concurrency like managing a busy editorial office where writers are constantly submitting draft revisions while editors are performing various types of review (quick fact-checks, detailed copy editing, comprehensive content analysis) that must all work with consistent versions of each document.

The primary concurrency challenge involves the **document change flow** where `didChange` notifications arrive asynchronously while completion, hover, and other feature requests are being processed. The `DocumentManager` must coordinate access to document content and ensure that feature providers work with consistent document versions.

Concurrency Pattern	Coordination Mechanism	Race Condition Risk	Debugging Approach
Request processing	Per-request promises/futures	Overlapping requests accessing shared state	Log request timestamps and resource access patterns
Document updates	Version-based synchronization	Feature requests seeing partial updates	Track document version consistency across operations
Background analysis	Debounced task scheduling	Analysis results overwriting newer analysis	Monitor analysis task queuing and completion timing
Cache management	Read-write locks or atomic updates	Cache corruption from concurrent modifications	Implement cache operation logging and integrity checks

The debugging approach for concurrency issues involves implementing comprehensive request tracing that tracks the lifecycle of each request through the system, including queuing time, processing time, and resource

access patterns. Create load tests that simulate realistic concurrent usage patterns to expose race conditions and deadlock scenarios.

A particularly subtle concurrency issue involves the **debouncing** mechanism in diagnostic publishing. When the user types rapidly, multiple `didChange` notifications arrive in quick succession. The server should debounce analysis to avoid performing expensive operations for each keystroke, but must ensure that the final analysis reflects the most recent document content.

⚠ Pitfall: Document Version Race Conditions A common race condition occurs when a feature request begins processing with document version N, but a `didChange` notification updates the document to version N+1 before the feature request completes. The feature request returns results based on stale content, confusing the user and potentially corrupting editor state.

Implementation Guidance

The debugging infrastructure for an LSP server requires systematic logging, diagnostic endpoints, and testing tools that can simulate various failure conditions. The implementation should provide both real-time debugging capabilities during development and comprehensive error reporting for production deployments.

Technology Recommendations

Component	Simple Option	Advanced Option
Protocol Logging	Console logging with JSON formatting	Structured logging with winston/zap + log aggregation
Performance Monitoring	Basic timing with Date.now()	Detailed profiling with clinic.js/pprof + metrics collection
Memory Profiling	Process memory tracking	Heap snapshots + memory leak detection
Request Tracing	Request ID logging	Distributed tracing with correlation IDs
Error Reporting	File-based error logs	Centralized error reporting with context capture

Recommended File Structure

```
project-root/
  src/
    debug/
      protocol-logger.ts      ← JSON-RPC message logging and analysis
      performance-profiler.ts ← Latency tracking and memory monitoring
      diagnostic-server.ts    ← HTTP endpoint for runtime diagnostics
      test-client.ts          ← LSP client for protocol testing
    transport/
      stream-transport.ts    ← Enhanced with debugging hooks
    protocol/
      lsp-server.ts          ← Enhanced with error reporting
    utils/
      error-handling.ts      ← Centralized error classification and reporting
  test/
    integration/
      protocol-compliance.test.ts ← Protocol conformance tests
      performance.test.ts       ← Load and latency testing
      concurrency.test.ts      ← Race condition and deadlock tests
  logs/
  diagnostics/              ← Runtime log files
                            ← Memory dumps and performance reports
```

Protocol Debugging Infrastructure

The protocol debugging system should provide comprehensive visibility into JSON-RPC message flow, protocol state transitions, and error conditions. This infrastructure supports both development debugging and production monitoring.

```
/**  
  
 * ProtocolLogger captures and analyzes JSON-RPC message flow for debugging.  
 * Provides filtering, correlation, and analysis of protocol interactions.  
 */  
  
interface ProtocolLogger {  
  
    // TODO 1: Implement message logging with configurable detail levels  
  
    logMessage(direction: 'inbound' | 'outbound', message: JsonRpcMessage): void;  
  
  
    // TODO 2: Add correlation tracking to match requests with responses  
  
    correlateRequestResponse(requestId: string | number): MessageCorrelation;  
  
  
    // TODO 3: Implement protocol state tracking and validation  
  
    validateProtocolState(currentState: ServerState, message: JsonRpcMessage): ValidationResult;  
  
  
    // TODO 4: Create message filtering and search capabilities  
  
    filterMessages(criteria: MessageFilter): JsonRpcMessage[];  
  
  
    // TODO 5: Generate protocol compliance reports  
  
    generateComplianceReport(): ProtocolComplianceReport;  
  
}  
  
/**  
  
 * DiagnosticServer provides HTTP endpoints for runtime LSP server inspection.  
 * Allows external tools to query server state and performance metrics.  
 */  
  
interface DiagnosticServer {  
  
    // TODO 1: Implement server state inspection endpoints
```

```
getServerState(): ServerDiagnostics;

// TODO 2: Add document state inspection

getDocumentStates(): DocumentStateDiagnostics[];

// TODO 3: Provide performance metrics endpoints

getPerformanceMetrics(): PerformanceMetrics;

// TODO 4: Add cache inspection capabilities

getCacheStatistics(): CacheStatistics;

// TODO 5: Implement health check endpoints

healthCheck(): HealthStatus;

}
```

The protocol debugging system includes a `TestClient` implementation that can simulate various client behaviors and error conditions:

```
/**  
  
 * LspTestClient simulates language client behavior for protocol testing.  
 * Supports error injection, timing control, and capability simulation.  
 */  
  
class LspTestClient {  
  
    private serverProcess: ChildProcess;  
  
    private messageFramer: MessageFramer;  
  
    private responseHandlers: Map<string | number, Function>;  
  
    // TODO 1: Implement server process management with configurable startup  
  
    async startServer(command: string, args: string[]): Promise<void> {  
  
        // Start LSP server process and set up stdio communication  
  
        // Configure message framing and error handling  
  
        // Set up process cleanup and error monitoring  
  
    }  
  
    // TODO 2: Add capability-aware initialization with custom client capabilities  
  
    async initialize(clientCapabilities: Partial<ClientCapabilities>):  
Promise<ServerCapabilities> {  
  
        // Send initialize request with specified capabilities  
  
        // Wait for and validate server response  
  
        // Send initialized notification to complete handshake  
  
        // Return negotiated server capabilities for test validation  
  
    }  
  
    // TODO 3: Implement document lifecycle simulation  
  
    async simulateDocumentEditing(uri: string, editSequence: DocumentEdit[]): Promise<void> {  
  
        // Send didOpen with initial content  
  
        // Apply sequence of didChange operations with realistic timing  
    }  
}
```

```
// Validate server responses and state consistency

// Send didClose to complete lifecycle

}

// TODO 4: Add error injection capabilities for robustness testing

injectTransportError(errorType: TransportError, timing: ErrorTiming): void {

    // Simulate connection failures, malformed messages, or timing issues

    // Allow controlled error injection for testing error handling paths

}

// TODO 5: Implement concurrent request testing

async runConcurrencyTest(requestPattern: RequestPattern[]): Promise<ConcurrencyTestResult>

{
    // Execute multiple overlapping requests to test concurrent handling

    // Measure response times and validate result consistency

    // Detect race conditions and deadlock scenarios

}
}
```

Performance Debugging Tools

The performance debugging infrastructure tracks latency, memory usage, and resource utilization patterns to identify bottlenecks and optimization opportunities:

```
/**  
 * PerformanceProfiler tracks detailed timing and resource usage for LSP operations.  
 * Provides statistical analysis and bottleneck identification.  
 */  
  
class PerformanceProfiler {  
  
    private measurements: Map<string, number[]> = new Map();  
  
    private startTimes: Map<string, number> = new Map();  
  
    private memorySnapshots: MemorySnapshot[] = [];  
  
    // TODO 1: Implement operation timing with statistical analysis  
  
    startMeasurement(operation: string, metadata?: Record<string, any>): string {  
  
        // Generate unique measurement ID for correlation  
  
        // Record start timestamp with high precision  
  
        // Capture relevant context (document size, symbol count, etc.)  
  
        // Return measurement ID for endMeasurement call  
  
    }  
  
    endMeasurement(measurementId: string): void {  
  
        // Calculate elapsed time since startMeasurement  
  
        // Store measurement in appropriate operation bucket  
  
        // Update running statistics (mean, median, percentiles)  
  
        // Trigger alerts if measurement exceeds thresholds  
  
    }  
  
    // TODO 2: Add memory profiling with leak detection  
  
    captureMemorySnapshot(label: string): void {  
  
        // Record current memory usage by category  
  
        // Calculate memory delta since last snapshot  
  
        // Identify potential leak patterns in cache or document storage  
    }  
}
```

```
// Generate memory usage reports and recommendations

}

// TODO 3: Implement bottleneck analysis and recommendations

analyzeBottlenecks(timeRange: TimeRange): PerformanceAnalysis {

    // Identify operations with highest latency impact

    // Correlate performance with document characteristics

    // Generate optimization recommendations

    // Highlight regression patterns over time

}

// TODO 4: Add real-time performance monitoring

getRealtimeMetrics(): RealtimeMetrics {

    // Return current performance statistics

    // Include recent latency trends and memory usage

    // Provide performance health indicators

    // Enable external monitoring system integration

}

// TODO 5: Implement performance regression detection

detectRegressions(baseline: PerformanceBaseline): RegressionReport {

    // Compare current performance against baseline measurements

    // Identify statistically significant performance changes

    // Correlate regressions with recent code or data changes

    // Generate alerts for significant performance degradation

}

}
```

Language-Specific Debugging Hints

For TypeScript LSP server implementations:

- Use `console.time()` and `console.timeEnd()` for quick performance measurements during development
- Leverage `process.memoryUsage()` to track heap usage and detect memory leaks
- Use `util.inspect()` with custom depth settings for debugging complex object structures
- Implement custom JSON serialization for large objects to avoid blocking the event loop
- Use `setImmediate()` or `process.nextTick()` to yield control during long-running operations
- Configure TypeScript compiler with `sourceMap: true` for better stack traces in error logs
- Use `worker_threads` for CPU-intensive analysis operations that shouldn't block request processing

Milestone Checkpoints

Protocol Debugging Checkpoint: After implementing basic protocol debugging infrastructure, verify correct operation by:

1. Start the LSP server with debug logging enabled
2. Connect with a simple test client and perform initialization handshake
3. Check that all JSON-RPC messages are logged with correct framing and correlation
4. Intentionally send malformed messages and verify appropriate error responses
5. Monitor protocol state transitions during normal operation and shutdown

Expected output should include detailed message logs showing Content-Length headers, JSON payloads, request/response correlation, and protocol state changes.

Performance Debugging Checkpoint: After implementing performance monitoring, validate the system by:

1. Run completion requests on documents of varying sizes (1KB, 10KB, 100KB)
2. Verify that latency measurements are captured and statistical analysis is performed
3. Generate artificial load with concurrent requests and check for performance degradation
4. Monitor memory usage during extended editing sessions with multiple open documents
5. Review performance reports and verify bottleneck identification accuracy

Expected behavior includes sub-100ms completion latency for small documents, graceful performance degradation under load, and accurate identification of performance bottlenecks.

Concurrency Debugging Checkpoint: After implementing concurrency testing and monitoring:

1. Simulate rapid document changes while processing completion requests
2. Verify that document version consistency is maintained across concurrent operations
3. Test deadlock detection by creating artificial resource contention scenarios
4. Validate that diagnostic publishing doesn't interfere with interactive features
5. Check error handling for race conditions and resource conflicts

Expected results include consistent document state across all operations, no deadlocks under normal load, and graceful degradation when resource conflicts occur.

Future Extensions

Milestone(s): This section builds upon all completed milestones, providing a roadmap for additional features beyond Milestone 4 (Diagnostics & Code Actions). These extensions enhance the LSP server with advanced capabilities like workspace management, refactoring, semantic highlighting, and performance optimizations.

Think of the Language Server Protocol as a living ecosystem that continues to evolve. Once you've built the core foundation with document synchronization, language features, and diagnostics, the LSP specification offers a rich set of additional capabilities that can transform your basic language server into a comprehensive development environment. These extensions are like adding specialized tools to a workshop - each one serves specific developer needs and builds upon the existing infrastructure.

The beauty of LSP's design lies in its **progressive enhancement** philosophy. Clients automatically discover server capabilities during initialization, so you can add features incrementally without breaking compatibility with existing editors. This section outlines the most impactful extensions you can add to create a production-ready language server that rivals commercial IDE experiences.

Advanced Language Features

The core language features from Milestone 3 represent just the beginning of what modern developers expect from their development environment. Think of these advanced features as the difference between a basic text editor with syntax highlighting and a full-featured IDE that actively assists with code development.

Document Symbols and Workspace Symbols

Document symbols provide a hierarchical outline of a file's structure, enabling the editor's "outline view" or "symbol navigator" features. This is like creating a table of contents for source code, allowing developers to quickly navigate large files by jumping directly to functions, classes, or variables.

The `textDocument/documentSymbol` request expects a hierarchical response representing the symbol tree within a single file. Each symbol contains name, kind, range, and optional children, creating a nested structure that editors can display as an expandable tree view.

Symbol Information	Type	Description
name	string	The symbol's display name (function name, class name, etc.)
kind	SymbolKind	Classification like Function, Class, Variable, Method
range	Range	Full range including symbol body and implementation
selectionRange	Range	Range to highlight when symbol is selected (typically just the name)
children	Symbol[]	Nested symbols for hierarchical structures like classes containing methods
detail	string	Additional information like function signature or type
deprecated	boolean	Whether this symbol is marked as deprecated

Workspace symbols extend this concept across the entire project, implementing `workspace/symbol` requests that search for symbols by name across all files. This powers the "Go to Symbol in Workspace" command that lets developers quickly jump to any function or class regardless of which file contains it.

The key architectural challenge is maintaining an efficient **global symbol index** that can handle workspace-wide queries with sub-second response times. This requires building and maintaining a cross-file symbol database that updates incrementally as files change.

Decision: Symbol Index Architecture

- **Context:** Workspace symbol queries need to search across thousands of files with millisecond response times
- **Options Considered:**
 1. Linear search through all parsed files
 2. In-memory hash map indexed by symbol name
 3. Persistent symbol database with incremental updates
- **Decision:** In-memory hash map with persistent backing store
- **Rationale:** Hash map provides O(1) lookup for exact matches and efficient prefix filtering, while persistence enables fast server startup
- **Consequences:** Requires memory proportional to symbol count but enables instant workspace-wide symbol search

Semantic Highlighting

Traditional syntax highlighting uses regular expressions to colorize keywords and literals, but semantic highlighting leverages the language server's deep understanding of code semantics to provide contextually-aware coloring. Think of it as the difference between a spell-checker that highlights misspelled words versus one that understands grammar and can highlight different parts of speech.

Semantic highlighting can distinguish between local variables, parameters, class members, static methods, and deprecated symbols using different colors or styles. This visual richness helps developers quickly understand code structure and identify potential issues at a glance.

The `textDocument/semanticTokens/full` request returns an encoded array of tokens representing every semantically meaningful element in the document. The encoding uses delta compression to minimize payload size - each token specifies its offset from the previous token rather than absolute positions.

Token Encoding	Bits	Description
Delta Line	0-31	Line offset from previous token (0 = same line)
Delta Character	0-31	Character offset from previous token (or line start if different line)
Length	0-31	Token length in UTF-16 code units
Token Type	0-31	Index into predefined token type array (variable, function, class, etc.)
Token Modifiers	0-31	Bitfield of modifiers (static, readonly, deprecated, etc.)

The incremental variant `textDocument/semanticTokens/range` allows the server to provide semantic tokens for just the visible portion of large files, dramatically improving performance for documents with thousands of lines.

Inlay Hints

Inlay hints display additional information directly in the editor without cluttering the source code. Common examples include showing parameter names in function calls, inferred types for variables, or return types for functions. This is like having inline documentation that appears only when helpful.

Modern TypeScript editors show inlay hints for parameter names in function calls (`processUser(name: user, age: 25)`) and inferred types for variables (`const items: string[] = getItems()`). These hints dramatically improve code readability without requiring developers to modify their source code.

The `textDocument/inlayHint` request returns hints for a specific range, allowing editors to request hints only for visible code regions. Each hint specifies its position, label, and optional tooltip with detailed information.

Inlay Hint Properties	Type	Description
position	Position	Location where hint should be displayed
label	string or LabelPart[]	Text to display, optionally with multiple styled parts
kind	InlayHintKind	Parameter name, type annotation, or other classification
textEdits	TextEdit[]	Optional edits applied when hint is accepted/clicked
tooltip	string or MarkupContent	Detailed explanation shown on hover
paddingLeft	boolean	Whether to add space before hint
paddingRight	boolean	Whether to add space after hint

Code Actions and Refactoring

Code actions extend beyond simple quick fixes to include sophisticated refactoring operations that can transform code structure while preserving behavior. Think of refactoring as architectural renovation for code - restructuring the internal design without changing what the program does.

Advanced Refactoring Operations

Extract Method refactoring takes a selected block of code and converts it into a separate function, automatically determining parameter lists and return types. This operation requires sophisticated analysis to identify variable dependencies, ensure scope correctness, and generate appropriate function signatures.

The refactoring process involves several complex steps: analyzing variable usage to determine parameters, checking for return value requirements, ensuring the extracted code has no control flow that would break when moved, and generating a function name that follows the project's naming conventions.

Refactoring Operation	Complexity	Requirements
Rename Symbol	Medium	Cross-file symbol resolution, import/export tracking
Extract Method	High	Control flow analysis, variable dependency tracking
Inline Method/Variable	Medium	Usage analysis, scope validation
Move Symbol	High	Module dependency analysis, import statement updates
Change Method Signature	High	Call site identification, parameter mapping

Rename Symbol operations must handle complex scenarios like updating import statements, resolving symbol conflicts, and maintaining references across multiple files. The server needs to identify every reference to the symbol, validate that the new name doesn't conflict with existing symbols in each scope, and generate text edits that atomically update all references.

Source Actions

Source actions provide high-level code generation and organization features triggered through editor commands rather than diagnostic-specific quick fixes. These include "Add missing imports," "Remove unused imports," "Sort imports," and "Generate constructor from properties."

The `source.organizeImports` action analyzes import statements to remove unused imports, add missing imports for referenced symbols, and sort imports according to language conventions. This operation requires tracking symbol usage throughout the file and maintaining a database of available symbols from external modules.

Auto-import functionality watches for unresolved symbol references and suggests adding import statements. When a developer types a symbol name that isn't in scope, the server can offer code actions that add the appropriate import and automatically insert the symbol reference.

Workspace Management Features

As projects grow beyond single files, language servers need sophisticated workspace management capabilities. Think of workspace management as city planning for code - organizing and coordinating multiple files, dependencies, and configurations into a coherent development environment.

Multi-Root Workspace Support

Modern development often involves working with multiple related repositories or projects simultaneously. Multi-root workspaces allow a single language server instance to manage multiple project roots, each with its own configuration, dependencies, and build settings.

The `workspace/workspaceFolders` request allows clients to notify servers about workspace folder changes, enabling dynamic reconfiguration as developers add or remove project roots from their workspace.

Workspace Management	Responsibility	Implementation Challenge
Configuration Loading	Load settings per workspace root	Merge conflicting configurations
Dependency Resolution	Resolve imports across project boundaries	Handle version conflicts
Build Integration	Coordinate with build systems	Track generated files
File Watching	Monitor file changes across roots	Scale to thousands of files

Workspace Edit Capabilities

Complex refactoring operations often require changes across multiple files, creating new files, or deleting existing ones. Workspace edits provide atomic multi-file operations that editors can apply as single undoable units.

The `WorkspaceEdit` structure can include file operations (create, rename, delete) alongside text edits, enabling sophisticated refactoring scenarios like moving a class to a new file while updating all import statements.

Workspace Edit Operations	Type	Use Cases
textDocument changes	Map<URI, TextEdit[]>	Update existing file content
documentChanges	(TextDocumentEdit or FileOperation) []	Ordered sequence of file and text changes
changeAnnotations	Map<string, ChangeAnnotation>	Metadata explaining the purpose of changes

Performance and Scalability Extensions

As language servers handle larger projects and more complex analysis, performance optimizations become critical. Think of these optimizations as upgrading from a bicycle to a race car - the fundamental functionality remains the same, but the speed and capacity increase dramatically.

Incremental Analysis

Instead of re-analyzing entire files on every change, incremental analysis identifies the minimal set of affected code regions and updates only those portions. This is like updating a city map by redrawing only the changed neighborhoods rather than recreating the entire map.

Incremental parsing builds upon tree-sitter or similar parsing technologies that can efficiently update syntax trees when small portions of the input change. The language engine maintains dependency graphs between symbols and analysis results, enabling surgical updates when dependencies change.

Analysis Granularity	Update Scope	Performance Impact
Full File	Entire document	O(file size) - suitable for small files
Function Level	Modified function and its callers	O(function size + dependents)
Statement Level	Modified statements and affected symbols	O(change size + local dependencies)
Token Level	Individual syntax tree nodes	O(tokens changed) - maximum efficiency

Background Analysis

Background analysis moves expensive operations like cross-file dependency resolution and deep semantic checking to background threads, ensuring the editor remains responsive during intensive analysis periods.

The server maintains separate analysis queues for different priority levels: immediate (required for current cursor position), high priority (visible in editor), and background (full project analysis). This tiered approach ensures that interactive features like completion and hover remain fast even during comprehensive project analysis.

Analysis Priority	Trigger	Response Time Target
Immediate	Cursor position requests	<50ms
High Priority	Visible document changes	<200ms
Medium Priority	Open document changes	<1000ms
Background	Project-wide analysis	Best effort, interruptible

Distributed Analysis

For extremely large codebases, distributed analysis can spread work across multiple language server instances or dedicated analysis workers. This architecture is like having multiple research teams working on different aspects of the same project rather than a single person trying to understand everything.

The coordinator server receives LSP requests and delegates analysis tasks to specialized workers, aggregating results before responding to clients. This approach enables horizontal scaling for enterprise-scale codebases with millions of lines of code.

Integration Extensions

Modern development workflows integrate with numerous external tools and services. Language servers can serve as integration hubs, bridging between editors and the broader development ecosystem.

Build System Integration

Integration with build systems enables the language server to understand project dependencies, compilation flags, and generated code. This is like having the language server understand not just the source code but also how that code gets transformed into running programs.

Build system integration can provide accurate dependency resolution by reading build configuration files, understanding conditional compilation directives, and tracking generated source files. The server watches build configuration changes and updates its analysis accordingly.

Build System	Integration Method	Capabilities Provided
Bazel	BUILD file parsing	Precise dependency graph, build flags
CMake	CMakeLists.txt analysis	Include paths, preprocessor definitions
Maven/Gradle	POM/build.gradle parsing	Dependency resolution, multi-module projects
Package.json	Dependency analysis	Node.js module resolution, script definitions

Version Control Integration

Git integration enables features like blame information in hover tooltips, showing when each line was last modified and by whom. The server can also provide diff-aware analysis, highlighting recently changed code regions or focusing analysis on modified files.

Advanced version control features include showing symbol history (when was this function last changed), detecting merge conflicts in symbol definitions, and providing contextual information about recent changes that might affect the current code.

External Tool Integration

Language servers can integrate with external analysis tools like linters, formatters, and security scanners. Instead of running these tools separately, the server coordinates their execution and presents unified results through standard LSP diagnostic messages.

Tool integration can be configured per-workspace, allowing different projects to use different linting rules or formatting standards while maintaining a consistent editor experience. The server manages tool execution, result caching, and error reporting transparently.

Advanced Diagnostic Features

Beyond basic syntax and semantic errors, advanced diagnostic systems can provide sophisticated code quality analysis, performance suggestions, and architectural guidance.

Multi-Level Diagnostics

Advanced diagnostic systems provide multiple severity levels beyond the standard Error/Warning/Information/Hint categories. Code quality diagnostics might include categories like Performance, Security, Maintainability, and Style, each with configurable severity levels.

Diagnostic aggregation combines results from multiple analysis passes, presenting related issues as grouped diagnostics rather than overwhelming developers with dozens of individual problems. For example, unused variable diagnostics can be grouped by function or class scope.

Diagnostic Category	Examples	Typical Severity
Correctness	Type errors, null dereferencing	Error
Performance	Inefficient algorithms, memory leaks	Warning
Security	SQL injection, XSS vulnerabilities	Warning/Error
Style	Naming conventions, code formatting	Information
Architecture	Circular dependencies, layer violations	Warning

Contextual Diagnostics

Contextual diagnostics adapt their analysis based on the current development context. For example, strict type checking might be relaxed in test files, or performance diagnostics might be more aggressive in hot code paths identified through profiling data.

The diagnostic engine can integrate with external context sources like test coverage reports, performance profiles, or deployment configurations to provide more targeted and relevant analysis results.

Custom Protocol Extensions

The LSP specification provides extension mechanisms for language-specific or tool-specific features that don't fit standard protocol methods. Custom extensions enable specialized functionality while maintaining compatibility with standard LSP clients.

Language-Specific Extensions

Language servers can define custom request/notification methods using vendor-specific namespaces. For example, a Rust language server might provide `rust-analyzer/syntaxTree` requests for displaying syntax tree information, or a TypeScript server might offer `typescript/organizeImports` with language-specific import sorting rules.

Custom extensions should follow LSP conventions for naming (vendor/feature format) and capability negotiation (advertised in server capabilities). This ensures that clients can gracefully handle servers that support different extension sets.

Extension Type	Namespace Format	Example
Vendor-Specific	<code>vendor/feature</code>	<code>rust-analyzer/inlayHints</code>
Experimental	<code>experimental/feature</code>	<code>experimental/ast-viewer</code>
Language-Specific	<code>language/feature</code>	<code>typescript/organizeImports</code>

Implementation Guidance

Building these extensions requires careful planning and incremental development. The key is to prioritize extensions based on user impact and implementation complexity, starting with high-value, low-complexity features before tackling more ambitious capabilities.

Technology Recommendations

Extension Category	Simple Option	Advanced Option
Symbol Indexing	In-memory Map<string, Symbol[]>	SQLite database with FTS
Background Analysis	Node.js Worker Threads	Separate process pool
Build Integration	Static config file parsing	Dynamic build system API
Version Control	Git command-line integration	LibGit2 bindings

Recommended Extension Implementation Order

The most effective approach is to implement extensions in order of user impact and technical dependency. Start with document symbols since they build directly on existing AST infrastructure, then progress to workspace symbols, semantic highlighting, and finally advanced refactoring operations.

```
src/
└── extensions/
    ├── symbols/
    │   ├── document-symbols.ts      ← Start here
    │   ├── workspace-symbols.ts    ← Requires symbol indexing
    │   └── symbol-index.ts        ← Shared infrastructure
    ├── highlighting/
    │   ├── semantic-tokens.ts    ← Build on AST analysis
    │   └── token-encoder.ts       ← Delta compression utilities
    ├── refactoring/
    │   ├── extract-method.ts     ← Complex control flow analysis
    │   ├── rename-symbol.ts      ← Cross-file reference tracking
    │   └── workspace-edits.ts    ← Multi-file operation support
    └── diagnostics/
        ├── advanced-diagnostics.ts ← Multi-level error reporting
        └── context-aware.ts        ← Contextual analysis
└── core/
    └── language-engine.ts        ← Enhanced with extension hooks
```

Symbol Index Infrastructure

The workspace symbol functionality requires a robust symbol indexing system that can efficiently handle tens of thousands of symbols across large codebases.

```
// Complete symbol indexing infrastructure

export interface SymbolIndexEntry {

    name: string;

    kind: SymbolKind;

    containerName?: string;

    location: Location;

    score: number; // For ranking search results

}

export class WorkspaceSymbolIndex {

    private symbolsByName = new Map<string, SymbolIndexEntry[]>();

    private symbolsByFile = new Map<string, SymbolIndexEntry[]>();

    // TODO: Implement addDocument(uri: string, symbols: DocumentSymbol[])

    // TODO: Implement removeDocument(uri: string)

    // TODO: Implement search(query: string, limit: number): SymbolIndexEntry[]

    // TODO: Implement fuzzyMatch(query: string, symbolName: string): number

    // TODO: Implement rankResults(results: SymbolIndexEntry[], query: string)

}
```

Semantic Token Provider

Semantic highlighting requires efficient token generation that can handle large documents with thousands of tokens.

```
export class SemanticTokenProvider {
```

TYPESCRIPT

```
    private tokenTypes = [
        'namespace', 'type', 'class', 'enum', 'interface', 'struct',
        'typeParameter', 'parameter', 'variable', 'property', 'enumMember',
        'function', 'method', 'macro', 'keyword', 'modifier', 'comment',
        'string', 'number', 'regexp', 'operator'
    ];

    private tokenModifiers = [
        'declaration', 'definition', 'readonly', 'static', 'deprecated',
        'abstract', 'async', 'modification', 'documentation', 'defaultLibrary'
    ];

    public async provideSemanticTokens(uri: string): Promise<SemanticTokens> {
        // TODO: Get AST and symbol table for document
        // TODO: Walk AST nodes and classify each token
        // TODO: Encode tokens using delta compression
        // TODO: Return SemanticTokens with encoded data array
    }

    private encodeTokens(tokens: Array<{line: number, char: number, length: number, type: number, modifiers: number}>): number[] {
        // TODO: Implement delta compression encoding
        // TODO: Each token encodes offset from previous token
        // TODO: Return flat array of [deltaLine, deltaChar, length, tokenType, modifiers]
    }
}
```

Milestone Checkpoint: Document Symbols

After implementing document symbols, verify the extension works correctly:

1. **Symbol Hierarchy:** Open a source file with nested structures (classes containing methods, modules containing functions). The editor's outline view should display a hierarchical tree matching your code structure.
2. **Symbol Navigation:** Click on symbols in the outline view - the editor should jump to the correct location and highlight the symbol name (`selectionRange`), not the entire definition.
3. **Symbol Kinds:** Different symbol types should display appropriate icons in the outline view (function icons for methods, class icons for types, etc.).
4. **Performance:** Document symbol requests should complete within 100ms for files up to 1000 lines. Test with progressively larger files to verify performance remains acceptable.

Debugging Tips for Extensions

| Symptom | Likely Cause | How to Diagnose | Fix | ---|---|---| | Outline view empty | DocumentSymbol response malformed | Check LSP trace logs for response structure | Verify hierarchical Symbol[] format, not flat SymbolInformation[] | | Wrong navigation target | Incorrect selectionRange | Compare range vs selectionRange in response | Set selectionRange to symbol name only, range to full definition | | Missing symbol icons | Wrong SymbolKind values | Check kind numbers against LSP specification | Use correct SymbolKind enum values (1=File, 6=Method, 5=Class, etc.) | | Workspace symbols timeout | Index too slow for large projects | Profile symbol search performance | Implement trie-based search or database backing |

The future of language servers lies in these sophisticated extensions that transform simple text editors into powerful development environments. Each extension builds upon the solid foundation established in the core milestones, creating a comprehensive developer experience that rivals traditional IDEs while maintaining the flexibility and performance advantages of the Language Server Protocol architecture.

Glossary

Milestone(s): This section provides essential terminology definitions for all milestones, particularly supporting Milestone 1 (JSON-RPC & Initialization) with protocol and transport concepts, Milestone 2 (Document Synchronization) with synchronization terminology, Milestone 3 (Language Features) with symbol analysis terms, and Milestone 4 (Diagnostics & Code Actions) with error handling definitions.

Think of this glossary as your technical dictionary for LSP development - like having a knowledgeable colleague nearby who can quickly explain any unfamiliar term or concept you encounter while building your language server. Each definition provides not just the "what" but also the "why it matters" context that helps you understand how the concept fits into the larger LSP ecosystem.

Core LSP Terminology

The Language Server Protocol ecosystem has evolved its own vocabulary that combines concepts from distributed systems, compiler design, and IDE development. Understanding these terms precisely is crucial because they represent specific technical concepts with well-defined behaviors, not just general software engineering terms.

Term	Category	Definition	Why It Matters
LSP	Protocol	Language Server Protocol - a standardized JSON-RPC based protocol that enables communication between development tools (clients) and language intelligence servers	Solves the $N \times M$ integration problem by providing one standard interface instead of requiring each editor to implement custom support for every language
language server	Architecture	A separate process that provides language analysis capabilities (completion, diagnostics, hover info) to development tools via the LSP protocol	Enables language intelligence to be shared across multiple editors and allows the server to use different technologies than the client
language client	Architecture	A development tool (editor, IDE) that connects to language servers to provide enhanced editing features for specific programming languages	Responsible for UI presentation and user interaction while delegating language analysis to the server
capability negotiation	Protocol	The process during LSP initialization where client and server exchange information about which features each supports	Enables progressive enhancement - newer features work when both sides support them, but the connection remains functional with older clients/servers
$N \times M$ integration problem	Architecture	The exponential complexity that arises when N editors each need custom integrations with M programming languages, resulting in $N \times M$ different implementations	LSP reduces this to $N+M$ by providing a standard protocol that each editor and language implements once

JSON-RPC Communication

JSON-RPC provides the foundation for LSP communication, but LSP adds specific conventions and requirements on top of the base JSON-RPC specification. These terms reflect how LSP adapts JSON-RPC for language server use cases.

Term	Category	Definition	Why It Matters
JSON-RPC	Protocol	A stateless, light-weight remote procedure call protocol using JSON for data exchange, serving as the transport layer for LSP	Provides structured request/response communication with error handling, enabling reliable client-server interaction
Content-Length framing	Transport	HTTP-style message boundary detection using Content-Length headers followed by CRLF CRLF sequence before the JSON payload	Solves the message boundary problem when multiple JSON-RPC messages are sent over a continuous byte stream
message framing	Transport	The process of adding Content-Length headers to JSON-RPC messages to enable proper parsing from a continuous byte stream	Essential for reliable communication over stdin/stdout pipes where message boundaries aren't naturally defined
message dispatching	Transport	Routing incoming JSON-RPC messages to appropriate handler functions based on the method name field	Enables clean separation of concerns by allowing different handlers to focus on specific LSP capabilities
bidirectional communication	Protocol	Both LSP client and server can initiate requests to each other, not just client sending requests to server	Allows servers to proactively send diagnostics, request client actions, and provide real-time updates
streaming parser	Implementation	A parser that processes JSON-RPC data incrementally as it arrives over the transport, handling partial messages gracefully	Necessary because stdin/stdout may deliver message data in arbitrary chunks that don't align with message boundaries
buffer management	Implementation	Accumulating partial message data across multiple read operations until complete messages can be extracted	Handles the reality that operating system I/O may split messages across multiple read() calls
handler registration	Implementation	The process of mapping LSP method names (like "textDocument/completion") to specific handler functions	Provides clean extensibility and separation of concerns by isolating each LSP feature in its own handler

Document Synchronization

Document synchronization represents one of the most critical aspects of LSP implementation because it maintains the shared understanding of source code state between client and server. These terms describe the various strategies and mechanisms involved.

Term	Category	Definition	Why It Matters
document synchronization	Protocol	The process of keeping the language server's understanding of document content synchronized with the editor's current state	Foundation for all language features - if the server has stale content, all analysis results will be incorrect
incremental synchronization	Strategy	Sending only the changed portions of a document (as range-based edits) rather than the entire document content	Dramatically improves performance for large files by minimizing data transfer and enabling targeted re-analysis
version tracking	Concurrency	Using monotonically increasing version numbers to prevent race conditions and ensure document changes are applied in the correct order	Prevents corrupted document state when multiple changes arrive rapidly or out of order
position conversion	Encoding	Converting between LSP's line/character coordinates (UTF-16 code units) and internal byte offset positions used by parsers	Critical for correctness because LSP positions must account for multi-byte Unicode characters
UTF-16 code units	Encoding	The character encoding standard used by LSP for all position calculations, where some Unicode characters require multiple code units	Must be handled correctly or position calculations will be wrong for files containing emoji, accented characters, or other non-ASCII content
snapshot isolation	Concurrency	Maintaining immutable document versions so that long-running analysis operations work with stable content even as the document changes	Prevents analysis corruption when users continue typing while diagnostics or other features are still computing
change application	Algorithm	The process of applying range-based text edits to document content while maintaining correct character offsets	Complex due to UTF-16 encoding and the need to handle multiple overlapping or adjacent changes
full synchronization	Strategy	Sending the complete document content on every change rather than just the modified portions	Simpler to implement correctly but inefficient for large files - useful as a fallback when incremental sync fails

Protocol State Management

LSP servers must carefully manage their lifecycle and protocol state to ensure reliable operation and clean resource management. These terms describe the state machine and transitions involved.

Term	Category	Definition	Why It Matters
initialization handshake	Protocol	The required sequence of initialize request followed by initialized notification that establishes the LSP connection	Must complete successfully before any language features can be used - defines the contract between client and server
server lifecycle	State Management	The sequence of states an LSP server transitions through: Uninitialized → Initializing → Initialized → Shutdown → Exit	Ensures proper resource allocation and cleanup while preventing requests from being processed in invalid states
protocol state machine	Implementation	A finite state machine that tracks the server's current protocol state and validates legal transitions	Prevents protocol violations and ensures the server behaves correctly even when clients send unexpected message sequences
progressive enhancement	Design	Features work with older clients through capability checking, while newer features are enabled when both sides support them	Maintains backward compatibility while allowing the protocol to evolve and add new capabilities over time

Language Analysis Engine

The language analysis engine represents the heart of language server functionality, transforming source code into structured information that powers IDE features. These terms describe the core concepts and data structures involved.

Term	Category	Definition	Why It Matters
AST	Data Structure	Abstract Syntax Tree - a hierarchical representation of parsed source code that captures syntactic structure while abstracting away formatting details	Provides the structured foundation for all semantic analysis, symbol resolution, and code transformation operations
symbol table	Data Structure	A mapping from identifier names to their declarations, including type information, scope, and source location	Enables symbol resolution, completion, hover information, and go-to-definition by tracking what each identifier refers to
scope	Language Semantics	A visibility boundary that determines which symbol declarations are accessible from a given location in the source code	Critical for correct symbol resolution - the same name can refer to different symbols in different scopes
symbol resolution	Algorithm	The process of determining which declaration an identifier reference points to by searching through enclosing scopes	Fundamental to most language features - must correctly handle shadowing, imports, and complex scoping rules
incremental parsing	Performance	Parsing only the changed portions of source code rather than re-parsing entire files on every edit	Essential for performance in large codebases - enables real-time analysis by avoiding expensive full re-parsing
semantic analysis	Language Processing	Analysis that goes beyond syntax to understand meaning, including type checking, symbol resolution, and error detection	Provides the deep language understanding needed for advanced IDE features like intelligent completion and refactoring
cache invalidation	Performance	The process of removing or updating cached analysis results when the underlying source code changes	Critical for correctness - stale cached results lead to wrong completion suggestions and missed errors
scope hierarchy	Data Structure	The nested tree structure of scopes in a program, where inner scopes can access symbols from outer scopes	Models language scoping rules and enables correct symbol lookup by walking up the scope chain

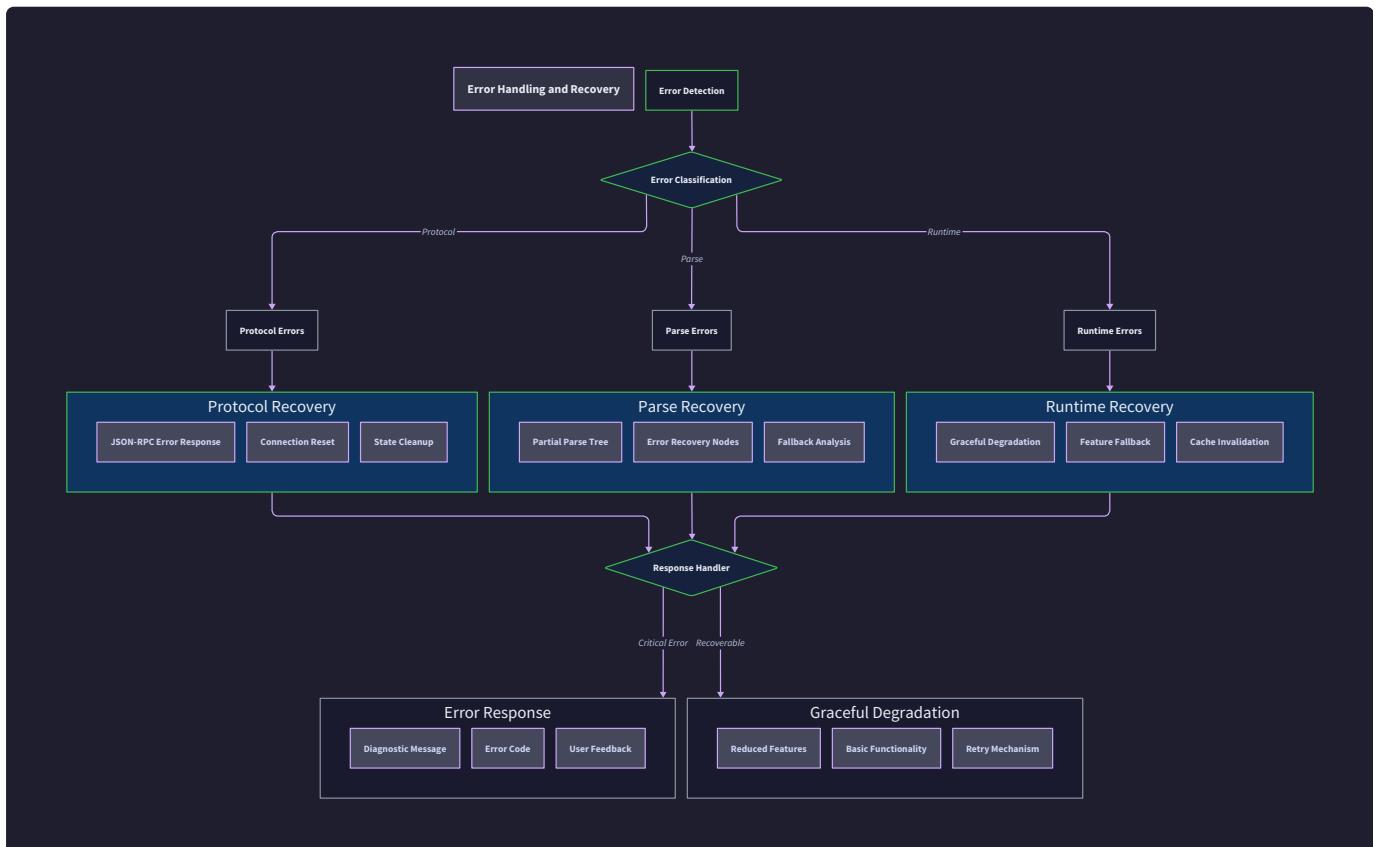
IDE Feature Implementation

LSP servers provide specific IDE capabilities through feature providers that analyze code and generate appropriate responses. These terms describe the various features and their implementation approaches.

Term	Category	Definition	Why It Matters
contextual filtering	Algorithm	The process of narrowing down from all possible completions to only those relevant at the current cursor position	Essential for usable completion - showing 10 relevant items is much more helpful than showing 1000 irrelevant ones
MarkupContent	Data Structure	LSP structure for returning rich formatted text (markdown or plaintext) in hover responses and completion documentation	Enables rich documentation display with formatting, code samples, and links in IDE tooltips
diagnostic severity	Classification	Error level categorization (Error, Warning, Information, Hint) that helps editors prioritize and display issues appropriately	Allows users to focus on critical errors while still seeing helpful suggestions and style recommendations
cross-file resolution	Algorithm	Following import statements and module references to resolve symbols defined in other files	Necessary for accurate analysis in modular codebases where most symbols are defined elsewhere

Error Handling and Recovery

Robust error handling distinguishes production-quality language servers from simple prototypes. These terms describe the various failure modes and recovery strategies required for reliable operation.



Term	Category	Definition	Why It Matters
error recovery	Strategy	Graceful handling of various failure modes without corrupting server state or crashing the process	Maintains server availability even when individual requests fail - essential for production use
protocol-level errors	Error Category	Failures in JSON-RPC communication such as malformed messages, unknown methods, or parameter validation errors	Must be handled according to JSON-RPC specification to maintain protocol compliance
parse error recovery	Algorithm	Strategies for continuing syntax analysis despite encountering invalid code, producing partial but useful results	Enables IDE features to work even in files with syntax errors, which is common during active development
capability degradation	Strategy	Temporarily disabling specific features when they repeatedly fail, while maintaining basic functionality	Prevents cascading failures from making the entire server unusable
partial AST construction	Implementation	Building incomplete but usable syntax trees when full parsing fails due to syntax errors	Allows symbol extraction and basic analysis even from syntactically invalid code
graceful feature degradation	Design	Maintaining reduced functionality when full features fail, rather than complete failure	Better user experience - some help is better than no help when things go wrong
panic mode recovery	Parsing	A parser recovery strategy that skips to known synchronization points (like statement boundaries) after encountering errors	Allows parsing to continue and find additional errors rather than stopping at the first syntax error

Testing and Debugging

Building reliable LSP servers requires comprehensive testing strategies and debugging approaches. These terms describe the tools and techniques for ensuring correctness.

Term	Category	Definition	Why It Matters
protocol compliance	Quality Assurance	Adherence to LSP specification requirements for message formats, state transitions, and feature behavior	Ensures compatibility with existing editors and tools that implement the LSP client specification
milestone checkpoints	Development Process	Verification procedures after each development phase to ensure correct implementation before proceeding	Prevents accumulated technical debt and makes debugging easier by catching issues early
integration testing	Testing Strategy	Testing the complete LSP server with real clients and realistic scenarios rather than just unit testing individual components	Catches issues that only appear in full client-server interaction, such as timing problems and protocol edge cases
protocol debugging	Debugging	Techniques for diagnosing JSON-RPC communication issues by logging and analyzing message exchanges	Essential for troubleshooting client-server communication problems and protocol compliance issues
performance debugging	Optimization	Identifying and fixing responsiveness issues that affect user experience in interactive editing scenarios	Critical for user adoption - slow language servers create frustrating editing experiences

Advanced Features and Extensions

Beyond basic language features, LSP supports advanced capabilities that enable sophisticated development tools. These terms describe the extended feature set available in modern language servers.

Term	Category	Definition	Why It Matters
symbol index	Data Structure	A searchable database of all symbols in a workspace, enabling fast workspace-wide symbol search and navigation	Powers features like "go to symbol in workspace" and enables analysis across large codebases
semantic highlighting	Feature	Context-aware syntax coloring that uses language analysis to provide more accurate and informative color coding	Provides better visual feedback than regex-based syntax highlighting by understanding semantic roles
inlay hints	Feature	Inline information display (like inferred types or parameter names) that appears in the editor without modifying source code	Enhances code readability by showing implicit information that aids understanding
workspace edit	Data Structure	An atomic operation that can modify multiple files with full undo support, used for refactoring operations	Enables safe large-scale code transformations like renaming across files

Implementation Guidance

Building an LSP server requires understanding not just the protocol specification, but also the practical implementation patterns and architectural decisions that lead to maintainable, performant servers.

Essential TypeScript Dependencies

For TypeScript LSP server development, you'll need these core dependencies:

Package	Purpose	Installation
<code>vscode-languageserver</code>	LSP server framework and protocol types	<code>npm install vscode-languageserver</code>
<code>vscode-languageserver-textdocument</code>	Document synchronization utilities	<code>npm install vscode-languageserver-textdocument</code>
<code>vscode-uri</code>	URI manipulation utilities compatible with LSP	<code>npm install vscode-uri</code>

Terminology Usage Patterns

When implementing your LSP server, use these specific terms consistently in your code and documentation:

Protocol Layer Terms:

- Always refer to "JSON-RPC messages" rather than just "messages" when discussing transport
- Use "Content-Length framing" when describing the header parsing mechanism
- Refer to "capability negotiation" for the feature agreement process, not "feature detection"
- Call the startup sequence "initialization handshake" to distinguish it from general initialization

Document Management Terms:

- Use "document synchronization" for the overall process, "incremental sync" vs "full sync" for specific strategies
- Refer to "version tracking" for the mechanism that prevents race conditions
- Use "position conversion" when discussing the transformation between line/character and byte offsets
- Call document change processing "change application" to emphasize it's an algorithmic process

Analysis Engine Terms:

- Use "symbol resolution" for the process of finding declarations, not "symbol lookup"
- Refer to "scope hierarchy" when discussing nested visibility boundaries
- Use "semantic analysis" for meaning-based analysis beyond syntax
- Call parser optimization "incremental parsing" rather than "partial parsing"

Error Handling Terms:

- Distinguish between "protocol-level errors" (JSON-RPC issues) and "parse error recovery" (syntax issues)

- Use "capability degradation" for the strategy of disabling failing features
- Refer to "graceful feature degradation" for maintaining reduced functionality

Common Implementation Mistakes

Understanding these pitfalls helps avoid the most frequent errors in LSP server development:

⚠ Pitfall: Incorrect UTF-16 Position Handling Many developers assume LSP positions work like typical line/column coordinates, but LSP uses UTF-16 code units. Characters like emoji (🚀) or accented letters (café) require multiple UTF-16 code units, so naive character counting produces wrong positions.

Why it's wrong: Position mismatches cause features to trigger at wrong locations, leading to incorrect completion suggestions and broken go-to-definition.

How to fix: Always use proper UTF-16 conversion functions when translating between LSP positions and internal byte offsets. The `vscode-languageserver-textdocument` package provides correct implementations.

⚠ Pitfall: Missing Message Boundary Detection Developers often assume each `stdin.read()` call returns exactly one complete JSON-RPC message, but the operating system can split messages across multiple reads or combine multiple messages in one read.

Why it's wrong: Leads to parsing errors when messages are fragmented or concatenated, causing intermittent connection failures.

How to fix: Always implement proper Content-Length header parsing and buffer incomplete messages until you have the full payload.

⚠ Pitfall: Blocking the Main Thread with Analysis Performing expensive parsing or semantic analysis synchronously in request handlers blocks the server from processing other requests, making the editor unresponsive.

Why it's wrong: Users experience laggy typing and delayed responses to other IDE features while analysis runs.

How to fix: Use asynchronous processing for expensive operations, implement analysis debouncing, and consider running heavy computation in worker threads.

⚠ Pitfall: Forgetting Protocol State Validation Accepting LSP requests before the initialization handshake completes or after shutdown begins violates the protocol specification and can cause undefined behavior.

Why it's wrong: Leads to protocol violations that break compatibility with some editors and can cause crashes or resource leaks.

How to fix: Implement a proper state machine that validates request legality based on current server state and rejects invalid requests with appropriate error codes.

⚠ Pitfall: Inconsistent Document Version Tracking Failing to properly validate document version numbers can lead to applying changes in the wrong order or to outdated document content.

Why it's wrong: Results in corrupted document state where the server's understanding of file contents doesn't match the editor, breaking all language features.

How to fix: Always validate version numbers are monotonically increasing and reject changes with unexpected versions, requesting a full document resync when versions get out of sync.

Development Workflow Recommendations

Follow this sequence when implementing LSP server components:

1. **Start with Transport Layer:** Get JSON-RPC message framing working correctly with proper Content-Length handling before moving to higher-level features.
2. **Implement Protocol State Machine:** Ensure initialization, shutdown, and state transitions work correctly before adding any language features.
3. **Build Document Synchronization:** Get document tracking working with both full and incremental sync modes before attempting any analysis.
4. **Add Basic Language Engine:** Implement simple parsing and symbol extraction before attempting complex features like completion.
5. **Implement Core Features:** Start with hover (simplest) and diagnostics, then add completion and go-to-definition.
6. **Add Error Handling:** Implement robust error recovery and graceful degradation after core features work.

This progression ensures you have a solid foundation at each step and can debug issues in isolation rather than fighting multiple broken components simultaneously.