Build Your Own Bundler: Design Document

Overview

This system transforms a collection of modular JavaScript/TypeScript files into optimized, production-ready bundles. The key architectural challenge is implementing static analysis on a graph of dependencies to perform tree shaking and code splitting, while accurately emulating complex Node.js module resolution rules and maintaining correct runtime semantics.

> 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):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

Modern web development has evolved from simple HTML pages with inline scripts to complex applications composed of hundreds, sometimes thousands, of modular JavaScript and TypeScript files. Each file declares dependencies on others through various import mechanisms, creating a tangled web of relationships that must be untangled and assembled into something a browser can efficiently download and execute. This is the core problem that bundlers solve: transforming a graph of interdependent modules into optimized, production-ready bundles.

Understanding why this problem exists requires examining the evolution of JavaScript module systems. In the early days, developers manually included `<script>` tags in HTML, managing dependencies through global variables—a brittle approach prone to naming collisions and load-order issues. **CommonJS** emerged with Node.js, bringing `require()` and `module.exports` for server-side modularity, but this synchronous API wasn't suitable for browsers where network latency dominates. **AMD (Asynchronous Module Definition)** attempted to solve this with async loading, but added boilerplate. Finally, **ES Modules (ESM)** arrived as an official JavaScript standard with `import` and `export` statements, offering static structure and native browser support. However, browsers still face performance penalties when loading hundreds of small files, and Node.js's rich module resolution algorithm (traversing `node_modules`, reading `package.json` fields) isn't native to browsers.

This fragmentation creates the **module management problem**: how do we take code written in multiple module formats (ESM, CommonJS) with complex dependency declarations, resolve all those declarations to actual files, combine them efficiently for network delivery, and eliminate dead code—all while preserving correct runtime semantics? Building a bundler from scratch is a complex but profoundly educational challenge that illuminates fundamental concepts in static analysis, graph algorithms, and JavaScript runtime behavior.

Mental Model: The Library Consolidation Project

Imagine you're a librarian tasked with consolidating a sprawling, disorganized research library. The library contains thousands of individual booklets (JavaScript modules), each referencing other booklets it depends on. Some booklets are written in different languages (ESM vs CommonJS), some reference booklets by vague nicknames ("lodash") rather than full titles, and many contain entire chapters that no one ever reads (unused exports). Your goal is to produce a small set of organized, minified volumes (bundles) that contain only the necessary content, properly cross-referenced, ready for efficient distribution.

The bundler performs this consolidation through a multi-stage process:

1. **Cataloging (Parsing)**: You examine each booklet's table of contents (AST) to note every external reference it makes.
2. **Address Resolution (Module Resolution)**: For each reference, you translate vague names like "helpers" into precise shelf locations like "Aisle 3, Shelf 2, Booklet 7" by following library filing rules.
3. **Map Drawing (Graph Building)**: You draw a dependency map showing how all booklets connect, revealing which are essential and which are orphaned.
4. **Weeding (Tree Shaking)**: You remove entire unused booklets and cross out unused chapters within kept booklets.
5. **Rebinding & Binding (Code Generation)**: You rewrite all internal references to use a consistent numbering system and add an index (runtime) that knows how to load each booklet in the correct order.

This mental model emphasizes that a bundler is fundamentally a **static analysis tool**—it examines code without executing it—followed by a **code transformation system** that rewrites modules to work in a consolidated environment.

The Module Management Problem

The challenge of module management breaks down into four core subproblems, each with its own complexities:

1. Module Format Divergence

JavaScript code exists in multiple module formats with different syntax and runtime behavior:

Format	Syntax	Loading	Key Characteristics
ES Modules (ESM)	`import x from './y'`, `export const z = 1`	Static, asynchronous in browsers	Static structure enables tree shaking; has live bindings (exports are live references)
CommonJS	`const x = require('./y')`, `module.exports = z`	Synchronous	Dynamic requires can be conditional; exports are copies at require time
Hybrid/Universal	Mix of both, often via build tools	Varies	Packages may have both `.mjs` and `.cjs` files or use `package.json` `"type": "module"`

A bundler must understand both syntaxes and bridge their semantic differences. For example, ESM's `import` statements are hoisted and can only appear at the top level, while CommonJS's `require()` can be called anywhere, even conditionally. This affects static analysis: dynamic `require()` calls may only be resolvable at runtime.

2. Module Resolution Complexity

When a module writes `import lodash from 'lodash'`, how does the bundler find the actual file? Node.js has a sophisticated resolution algorithm that:

- Checks relative paths (`./foo`, `../bar`)
- Traverses up directories looking for `node_modules`
- Consults `package.json` fields (`main`, `module`, `exports`)
- Handles extensions (`.js`, `.ts`, `.json`) and directory index files

This algorithm must be replicated precisely for the bundler to work with the existing npm ecosystem. The `exports` field in `package.json` adds further complexity with **conditional exports** (different paths for different environments) and **subpath patterns**.

3. Scope Isolation and Runtime Semantics

When modules are concatenated into a single file, their top-level variables would normally collide in the global scope. Each module must be **wrapped** in its own function scope to prevent this. Moreover, the runtime must emulate module loading semantics:

- **Caching**: When a module is imported multiple times, the same instance should be returned.
- **Circular Dependencies**: When Module A imports Module B, and Module B imports Module A, the runtime must handle this without infinite recursion or undefined values.
- **Live Bindings** (ESM): When Module A exports a mutable variable, and Module B imports it, changes in Module A should be visible in Module B. CommonJS exports copies, not references.

4. Optimization Through Static Analysis

Simply concatenating all modules would include every line of code from every dependency, including unused exports. **Tree shaking** (dead code elimination) uses the static structure of ESM to determine which exports are actually imported and transitively used, removing the rest. This requires:

- Building a complete import/export graph
- Tracking side effects (code that executes independent of exports, like polyfills)
- Respecting `package.json`'s `sideEffects: false` hint

Additionally, **code splitting** allows creating multiple bundles for different application routes, loaded on demand via dynamic `import()` calls.

> The critical insight is that a bundler operates in two phases: **static analysis** (understanding the module graph without executing code) and **code transformation** (rewriting modules to work in a bundled environment while preserving runtime behavior).

Existing Solutions

Several mature bundlers have emerged, each with different architectural philosophies and optimization strategies. Understanding their approaches informs our design decisions.

Bundler	Primary Architecture	Plugin System	Optimization Strategy	Key Differentiator
webpack	Configuration-driven, plugin-centric	Extensive, with tapable hooks	Chunk splitting, loader-based transformations	Extreme flexibility via loaders for all asset types; complex but powerful configuration
Rollup	ES module-first, tree shaking as core	Plugin API focused on hooks during bundle phases	Aggressive tree shaking via static analysis	Clean output with minimal runtime; excellent for library authors
esbuild	Go-written, parallelism-first	Limited plugin API (on-resolve, on-load)	Parallel parsing, bundling, and minification	Blazing speed (10-100x faster); simplicity over configurability
Parcel	Zero-configuration, asset pipeline	Plugin-based but aims for zero config	Multi-core processing, caching for incremental builds	Out-of-the-box experience with no configuration needed
Vite	Dev server with esbuild, prod with Rollup	Rollup-compatible plugin ecosystem	Unbundled dev server (ESM native), optimized prod bundles	Lightning-fast development via native ESM in browser during dev

Architecture Comparison:

- **webpack** uses a **tapable** event system where plugins hook into various compilation stages. It maintains a **dependency graph** where modules are connected via dependencies (imports/requires). Each module passes through loaders (transformers) before being added to chunks.
- **Rollup** uses a **plugin pipeline** that processes modules in stages: `resolveId`, `load`, `transform`, `renderChunk`. Its core innovation is **tree shaking via static analysis**—it builds an abstract syntax tree (AST) for each module, traces import/export relationships, and eliminates unused code before generating output.
- **esbuild** is written in Go and leverages parallelism and careful memory management. It parses, resolves, and links modules in parallel, avoiding JavaScript's single-threaded limitations. Its architecture prioritizes speed over extensibility.

Plugin Ecosystem Implications: A rich plugin ecosystem allows extending the bundler for non-JS assets (CSS, images) and custom transformations (TypeScript, Babel). However, plugins increase complexity and can make tree shaking harder (if plugins introduce non-analyzable side effects). Our educational bundler will prioritize core bundling concepts over plugin extensibility.

Optimization Trade-offs:

- **Tree shaking precision** vs **build speed**: Rollup's precise static analysis is slower than esbuild's faster but less aggressive approach.
- **Configuration complexity** vs **out-of-the-box behavior**: webpack's flexibility comes with configuration burden; Parcel's zero-config may not fit complex needs.

> **Decision: Learning-First Architecture**
>
> - **Context**: This project aims to teach bundler fundamentals, not compete with production tools.
> - **Options Considered**:
>   1. **Webpack-style plugin architecture**: Highly extensible but complex to implement.
>   2. **Rollup-style phased pipeline**: Clear separation of concerns, excellent for understanding stages.
>   3. **Esbuild-style speed-first**: Less educational about algorithm details.
> - **Decision**: Adopt a **Rollup-inspired phased pipeline** for clarity.
> - **Rationale**: The phased approach (parse → resolve → transform → generate) mirrors the logical steps of bundling and makes each component's responsibility clear. This aligns with our educational goals.
> - **Consequences**: Our bundler will be easier to understand and extend in a learning context, though it may not be as performant as parallel architectures.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
AST Parser	`@babel/parser` with TypeScript plugin	`acorn` + `acorn-walk` for smaller footprint
File System	Node.js `fs/promises` API	Caching layer for repeated reads
Path Resolution	Implement Node resolution algorithm manually	Use `resolve` npm package for spec compliance
Graph Data Structure	JavaScript `Map` and `Set` with adjacency lists	Optimized graph library for large projects
Code Generation	Template strings for runtime, `magic-string` for source maps	Custom code printer with AST transformations

Recommended File/Module Structure:

```
bundler/
├── package.json
├── src/
│   ├── index.ts                  # CLI entry point
│   ├── types/                    # TypeScript type definitions
│   │   └── module-node.ts
│   ├── parser/
│   │   ├── index.ts              # Main parser interface
│   │   ├── javascript-parser.ts  # AST parsing with Babel
│   │   └── dependency-extractor.ts # Extracts imports/exports from AST
│   ├── resolver/
│   │   ├── index.ts              # resolve(specifier, fromDir)
│   │   ├── node-resolver.ts      # Node.js resolution algorithm
│   │   └── package-json-reader.ts # Reads and caches package.json
│   ├── graph/
│   │   ├── index.ts              # ModuleGraph class
│   │   ├── builder.ts            # Builds graph from entry point
│   │   └── tree-shaker.ts        # Mark-and-sweep algorithm
│   ├── generator/
│   │   ├── index.ts              # generateBundle(graph)
│   │   ├── runtime.ts            # Runtime template
│   │   ├── module-wrapper.ts     # Wraps module code
│   │   └── source-map.ts         # Source map generation
│   └── utils/
│       ├── logger.ts             # Colored console output
│       └── fs-cache.ts           # File system caching
└── test/
    ├── fixtures/                 # Test projects
    └── unit/                     # Unit tests per component
```

Infrastructure Starter Code:

JAVASCRIPT

```javascript
// src/utils/fs-cache.ts
import { readFile } from 'fs/promises';
import { resolve } from 'path';

/**
 * Simple file system cache to avoid repeated disk reads
 */
export class FSCache {
  constructor() {
    this.cache = new Map();
  }

  async readFile(filePath) {
    const normalized = resolve(filePath);

    if (this.cache.has(normalized)) {
      return this.cache.get(normalized);
    }

    const content = await readFile(normalized, 'utf-8');
    this.cache.set(normalized, content);
    return content;
  }

  clear() {
    this.cache.clear();
  }
}

// src/utils/logger.ts
export const logger = {
  info: (msg) => console.log(`\x1b[36m[INFO]\x1b[0m ${msg}`),
  warn: (msg) => console.log(`\x1b[33m[WARN]\x1b[0m ${msg}`),
  error: (msg) => console.log(`\x1b[31m[ERROR]\x1b[0m ${msg}`),
  success: (msg) => console.log(`\x1b[32m[SUCCESS]\x1b[0m ${msg}`)
};
```

Core Logic Skeleton Code:

JAVASCRIPT

```javascript
// src/types/module-node.ts

/**
 * Represents a single module in the dependency graph
 */
export class ModuleNode {
  /**
   * @param {string} id - Unique identifier (usually absolute path)
   * @param {string} filePath - Absolute file system path
   */
  constructor(id, filePath) {
    this.id = id;
    this.filePath = filePath;
    this.dependencies = new Set(); // ModuleNode IDs this module imports
    this.dependents = new Set();   // ModuleNode IDs that import this module
    this.exports = new Set();      // Names of exports from this module
    this.imports = new Map();      // Map of import name -> source module ID
    this.ast = null;               // Parsed AST
    this.source = '';              // Original source code
    this.sideEffects = true;       // Whether module has side effects
    this.isEntry = false;          // Whether this is an entry point
  }
}


// src/graph/builder.ts

/**
 * Builds a module graph starting from entry points
 */
export class GraphBuilder {
  /**
   * @param {Object} options
   * @param {Function} options.resolve - Resolver function
   * @param {Function} options.parse - Parser function
   */
  constructor({ resolve, parse }) {
    this.resolve = resolve;
    this.parse = parse;
    this.modules = new Map(); // id -> ModuleNode
```

```
    this.fsCache = new FSCache();
  }

  /**
   * Builds the complete module graph from entry points
   * @param {string[]} entryPaths - Absolute paths to entry files
   * @returns {Promise<ModuleGraph>} The complete module graph
   */
  async build(entryPaths) {
    // TODO 1: Create ModuleNode for each entry path
    // TODO 2: For each entry, start BFS/DFS traversal
    // TODO 3: For each module, parse to get its dependencies
    // TODO 4: For each dependency, resolve to absolute path
    // TODO 5: Create ModuleNode for new dependencies, add edges
    // TODO 6: Continue until all transitive dependencies are discovered
    // TODO 7: Return a ModuleGraph instance containing all nodes
  }
}
```

Language-Specific Hints:

- Use `@babel/parser` with the `typescript` and `jsx` plugins to parse modern JavaScript/TypeScript
- Use `@babel/traverse` or a custom visitor to extract import/export statements from AST
- Use `path.resolve()` and `path.dirname()` for path manipulation
- Store module relationships in `Map` and `Set` for O(1) lookups
- Use `new Error().stack` or custom error classes for debugging

Milestone Checkpoint (Context Verification): After setting up the project structure, verify your environment:

```bash
# Install dependencies

npm init -y

npm install @babel/parser @babel/traverse @babel/types


# Create a test entry file

echo "import { foo } from './utils'; console.log(foo);" > test.js

echo "export const foo = 'bar';" > utils.js


# Run a simple test script to verify Babel parsing works

node -e "

const parser = require('@babel/parser');

const fs = require('fs');

const code = fs.readFileSync('test.js', 'utf-8');

const ast = parser.parse(code, { sourceType: 'module' });

console.log('AST parsed successfully:', ast.type);

"
```

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
"Cannot find module" error during parsing	Missing Babel plugins for syntax	Check parser configuration; ensure plugins match file extensions	Add necessary plugins (`typescript` , `jsx` , `decorators`)
Infinite loop in resolution	Circular symlinks or mis-handled node_modules traversal	Log each resolution step; check for repeated paths	Implement visited set to detect cycles
All exports removed after tree shaking	Side effects not properly detected	Check `package.json` `sideEffects` field; look for top-level function calls	Mark modules with side effects; preserve code with global assignments

Goals and Non-Goals

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section establishes the **scope boundaries** for our educational bundler project. Defining clear goals and non-goals is critical in any systems design process—it prevents **scope creep** while ensuring we focus on implementing the core concepts that make a bundler educational. Think of this as drawing the architectural blueprint for a house: we're specifying which rooms we'll build (goals) and which amenities we'll defer to future renovations (non-goals).

The bundler we're building is intentionally **pedagogical** rather than **production-ready**. Its primary purpose is to illuminate the fundamental algorithms and data structures behind tools like webpack and Rollup, not to compete with them in performance or feature completeness. Each goal corresponds to a key insight about how modern JavaScript tooling works, while each non-goal represents a deliberate trade-off to keep the project achievable while still covering the essential learning outcomes.

Goals

Our bundler must implement the **five foundational pillars** of modern JavaScript bundling. Each goal represents a core architectural capability that, when combined, produces a working bundling system capable of transforming modular code into optimized browser-ready bundles.

Goal	Description	Corresponding Milestone	Learning Outcome
ES Module Parsing	Parse JavaScript/TypeScript source files to extract their static dependency structure. This includes recognizing `import`, `export`, `import()`, and `require()` statements and building an accurate representation of the module's public interface.	Milestone 1	Understanding how static analysis enables dependency graph construction
Node.js Module Resolution	Implement the Node.js resolution algorithm to translate module specifiers (like `./utils` or `react`) into absolute file system paths. This includes handling `package.json` fields, `node_modules` traversal, and path resolution rules.	Milestone 2	Learning how module specifiers map to physical files in complex package ecosystems
Basic Runtime System	Generate a runtime module loader that executes modules in correct dependency order, handles scope isolation, and provides the `require`/`export` API that modules expect at execution time.	Milestone 3	Understanding how module isolation and dependency management work at runtime
Tree Shaking	Perform static analysis to eliminate dead code by tracing export usage across the module graph and removing unused exports and their associated implementation code.	Milestone 4	Learning how static analysis enables optimization beyond simple concatenation
Source Map Generation	Produce source maps in the standard v3 format that map positions in the bundled output back to original source files, enabling debugging of the original source code rather than the transformed bundle.	Milestone 3	Understanding how developer tooling integrates with build systems

Detailed Goal Breakdown:

ES Module Parsing: The bundler must correctly parse both **JavaScript** and **TypeScript** source files (via the TypeScript compiler or Babel) to build an Abstract Syntax Tree (AST). From this AST, it must extract:

- **Static import declarations** (`import x from './module'`)
- **Export declarations** (`export const x = 1`, `export { x } from './module'`)
- **Dynamic import expressions** (`import('./module')`) for code splitting points
- **CommonJS `require()` calls** for backward compatibility

The parser must produce a **complete dependency list** for each module, distinguishing between **static dependencies** (known at build time) and **dynamic dependencies** (potential code split points). This forms the foundation for constructing an accurate module dependency graph.

Node.js Module Resolution: The resolver must implement the **exact resolution algorithm** used by Node.js and bundlers, including:

- **Relative path resolution** (`./file.js`, `../directory/file.js`)
- **Absolute path resolution** (`/project/src/file.js`)
- **Node modules resolution** for bare specifiers (`react`, `lodash/get`)
- **Package.json field lookup** in priority order: `exports` > `module` > `main` > `browser`
- **File extension resolution** (`.js`, `.ts`, `.json`) and **directory index resolution** (`./directory/` → `./directory/index.js`)

This ensures the bundler works with real-world npm packages and follows community-standard resolution rules.

Basic Runtime System: The generated bundle must include a **minimal but functional runtime** that:

- **Wraps each module** in a function scope to prevent global namespace pollution
- **Maintains a module registry** that caches loaded modules
- **Provides `require()` and `export` functions** that modules can call
- **Handles circular dependencies** correctly (though this is challenging)
- **Executes modules in topological order** based on their dependencies

The runtime doesn't need to be production-optimized but must correctly implement the **module loading semantics** that developers expect.

Tree Shaking: The bundler must perform **static dead code elimination** by:

- **Tracing export usage** from entry points through the entire module graph
- **Marking "live" exports** that are imported somewhere in the application
- **Pruning "dead" exports** and their implementation code from the final bundle

- **Respecting `sideEffects` field** in `package.json` to safely remove entire modules
- **Preserving code with side effects** even if its exports are unused (e.g., polyfills)

This requires building a **complete usage graph** and understanding the difference between **pure modules** (safe to remove) and **modules with side effects** (must be preserved).

Source Map Generation: The bundler must produce **standard source maps** that:

- **Map bundled positions** to original source file, line, and column
- **Support multiple source files** contributing to a single bundle
- **Include original source content** for tools that need to display source code
- **Follow the Source Map v3 specification** for interoperability with browser devtools

> **Design Insight:** These five goals form a **minimum viable bundler** that teaches the core concepts without overwhelming complexity. Each goal builds upon the previous: parsing enables resolution, resolution enables graph building, graph building enables tree shaking, and all together enable code generation with source maps. This layered approach mirrors how production bundlers are architected.

Non-Goals

The following features are **explicitly out of scope** for this educational project. Including them would increase complexity without proportionally increasing educational value. Each represents a **deliberate exclusion** based on the principle of focusing on core algorithms rather than peripheral features.

Non-Goal	Reason for Exclusion	Alternative for Learning
Hot Module Replacement (HMR)	HMR requires complex runtime coordination, websocket connections, and state preservation that distracts from core bundling algorithms. It's a valuable production feature but represents an advanced optimization rather than a core concept.	Learners can implement a simple file watcher that triggers rebundling as a stretch goal.
Full Plugin Ecosystem	Production bundlers have extensive plugin systems for extensibility. Implementing a robust plugin architecture requires significant infrastructure for hooks, lifecycle events, and configuration management.	The design includes extension points where plugins could be added, but we won't implement the plugin API itself.
On-Disk Caching	Persistent caching (like webpack's filesystem cache) optimizes rebuild performance but adds complexity with cache invalidation, serialization, and versioning. It's a performance optimization, not a core bundling concept.	Learners can implement a simple in-memory cache for resolved modules as an optimization exercise.
Non-JS Asset Processing	Processing CSS, images, fonts, etc., requires loaders, transformers, and specialized handling that differs significantly from JavaScript module processing. It expands scope beyond the core module bundling problem.	The design can be extended with asset-specific transformers later, but they're not required for understanding module bundling fundamentals.
Production Optimizations	Minification, chunk hash naming, aggressive compression, and other production optimizations are important but represent applied optimizations on top of the core bundling algorithm.	Learners can integrate existing minifiers (like Terser) via CLI piping as a separate build step.
Multiple Output Formats	Supporting CommonJS, UMD, AMD, and SystemJS outputs requires different runtime wrappers and export strategies. ESM-to-ESM bundling is complex enough for educational purposes.	The bundler outputs a single self-contained IIFE bundle suitable for browser execution.
Type Checking	While we parse TypeScript, we won't implement type checking or emit type errors. Type checking is a separate compilation phase that doesn't directly relate to bundling algorithms.	Learners can run `tsc --noEmit` separately or integrate the TypeScript compiler API for type checking as an extension.
Monorepo Support	Advanced monorepo features like workspace resolution, hoisting, and cross-package references add significant complexity to module resolution.	The resolver handles standard `node_modules` resolution, which works for simple monorepo structures without special handling.

Rationale for Exclusions:

Hot Module Replacement (HMR) represents a **developer experience optimization** rather than a core bundling algorithm. Implementing HMR would require:

1. A websocket server for client-server communication
2. Runtime module update logic that patches application state
3. Complex dependency tracking for determining what to update
4. State preservation across module replacements

These concepts, while valuable, distract from understanding how modules are **statically analyzed**, **resolved**, and **concatenated**—the primary educational objectives.

Plugin Systems in production bundlers (like webpack's 100+ hook points) enable ecosystem extensibility but require:

- A well-defined plugin interface with lifecycle hooks
- Configuration schema validation
- Context sharing between plugins
- Error handling across plugin boundaries

Instead of building a plugin system, our design will have **clearly defined interfaces** between components (Parser, Resolver, Graph Builder, Generator) that could later be extended into a plugin system. This teaches the **separation of concerns** without implementing the full extensibility machinery.

On-Disk Caching addresses **performance concerns** rather than correctness. A production bundler might cache:

- Parsed ASTs
- Resolved module paths
- Transformed module code
- Dependency graphs

Implementing robust caching requires solving hard problems like:

- Cache key generation (content hashing, environment factors)
- Cache invalidation (when do dependencies change?)
- Serialization/deserialization of complex objects
- Cache versioning and migration

These are important engineering concerns but secondary to understanding the **fundamental algorithms**. Learners can add a simple `Map`-based in-memory cache as an exercise.

Non-JS Asset Processing fundamentally changes the problem domain. JavaScript modules have well-defined syntax and semantics; assets like CSS, images, and fonts require:

- Specialized parsers/transformers
- Different optimization strategies
- Unique runtime handling (CSS injection, image URLs)
- Loader chains with potentially async operations

By focusing exclusively on JavaScript/TypeScript, we maintain **conceptual coherence**. The module resolution, graph construction, and code generation algorithms we implement are directly applicable to any asset type that can be treated as a module, but implementing those transformers would triple the project scope.

> **Design Principle:** Every non-goal represents a **deliberate simplification** to maintain educational focus. Production bundlers like webpack are complex because they solve all these problems simultaneously; our educational bundler isolates the core algorithms so they can be understood independently. This follows the Unix philosophy: "Do one thing well."

Scope Boundaries in Practice

To make these boundaries concrete, consider what happens when our bundler encounters different scenarios:

✅ In Scope (Handled by Goals):

- `import { Component } from './Component.js'` → Relative path resolution to actual file
- `import React from 'react'` → Node modules resolution to `node_modules/react/index.js`
- `export const pi = 3.14` → Export extraction for tree shaking analysis

- `import('./DynamicModule.js')` → Marked as code split point (though we generate single bundle)
- `require('legacy-module')` → CommonJS interoperability via runtime wrapper
- `module.sideEffects = false` in `package.json` → Enables aggressive tree shaking

❌ **Out of Scope (Excluded by Non-Goals):**

- `import styles from './App.css'` → CSS modules require CSS parser and runtime injection
- WebSocket connection for HMR updates → Requires dev server infrastructure
- `cache: { type: 'filesystem' }` → Persistent caching layer
- `plugins: [new MyPlugin()]` → Plugin API implementation
- `output.libraryTarget: 'umd'` → Multiple output format support
- `tsc --noEmit` type checking → Separate compilation phase

The bundler will **fail gracefully** for out-of-scope features—either ignoring them (for CSS imports, treating them as external dependencies) or throwing clear error messages explaining the limitation.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
ES Module Parser	`@babel/parser` with plugin support	TypeScript Compiler API (more accurate for TS)
AST Traversal	`@babel/traverse` for Babel AST	Custom visitor pattern for learning
Module Resolution	Custom implementation of Node algorithm	`resolve` npm package (production-ready)
Source Map Generation	`source-map` library from Mozilla	Custom VLQ encoding for deeper understanding
File System Operations	Node.js `fs/promises` API	With caching layer for performance

Recommended Project Structure

```
build-your-own-bundler/
├── package.json
├── tsconfig.json              # TypeScript configuration
├── src/
│   ├── cli/                   # Command-line interface
│   │   └── index.ts           # CLI entry point
│   ├── bundler/               # Main bundling pipeline
│   │   └── index.ts           `BundleBuilder` class orchestrating the pipeline
│   ├── parser/                # Corresponds to Milestone 1
│   │   ├── index.ts           `Parser` class with `parseFile(filePath)`
│   │   ├── javascript-parser.ts
│   │   └── typescript-parser.ts
│   ├── resolver/              # Corresponds to Milestone 2
│   │   ├── index.ts           `Resolver` class with `resolve(specifier, fromDir)`
│   │   ├── node-resolver.ts   # Node modules resolution
│   │   └── package-json.ts    # Reads and interprets package.json fields
│   ├── graph/                 # Corresponds to Milestones 1, 3, 4
│   │   ├── index.ts           `ModuleGraph` class
│   │   ├── module-node.ts     `ModuleNode` type definition
│   │   ├── builder.ts         `GraphBuilder.build(entryPaths)`
│   │   └── shaker.ts          Tree shaking algorithms
│   ├── generator/             # Corresponds to Milestone 3
│   │   ├── index.ts           `Generator` class with `generateBundle(moduleGraph)`
│   │   ├── runtime.ts         # Runtime module loader template
│   │   └── source-map.ts      # Source map generation
│   └── utils/
│       ├── fs-cache.ts        `FSCache.readFile(filePath)` with caching
│       ├── logger.ts          # Logging utilities
│       └── errors.ts          # Custom error types
└── test/                      # Test fixtures and suites
    ├── fixtures/              # Test projects for integration tests
    │   ├── simple-app/
    │   ├── node-modules-app/
    │   └── tree-shake-app/
    └── unit/                  # Unit tests for each component
```

Infrastructure Starter Code

File System Cache Utility (Complete Implementation):

JAVASCRIPT

```javascript
// src/utils/fs-cache.ts

import fs from 'fs/promises';

import path from 'path';


/**

 * Simple file system cache with TTL (time-to-live) support.

 * Used to avoid repeated disk reads during bundling.

 */

export class FSCache {

  private cache = new Map<string, { content: string; timestamp: number }>();

  private ttl: number; // milliseconds


  constructor(ttl = 5000) {

    this.ttl = ttl;

  }


  /**

   * Read file with caching. Returns cached content if available and fresh.

   * @param filePath Absolute path to file

   * @returns Promise resolving to file content

   */

  async readFile(filePath: string): Promise<string> {

    const normalizedPath = path.normalize(filePath);

    const now = Date.now();



    // Check cache

    const cached = this.cache.get(normalizedPath);

    if (cached && (now - cached.timestamp) < this.ttl) {

      return cached.content;

    }



    // Read from disk

    try {

      const content = await fs.readFile(normalizedPath, 'utf-8');

      this.cache.set(normalizedPath, { content, timestamp: now });

      return content;

    } catch (error) {

      throw new Error(`Failed to read file ${normalizedPath}: ${error.message}`);
```

```
    }
  }

  /**
   * Invalidate cache for a specific file or clear all cache
   * @param filePath Optional path to invalidate, otherwise clears all
   */
  invalidate(filePath?: string) {
    if (filePath) {
      this.cache.delete(path.normalize(filePath));
    } else {
      this.cache.clear();
    }
  }
}

// Singleton instance for convenience
export const fsCache = new FSCache();
```

Logger Utility (Complete Implementation):

```javascript
// src/utils/logger.ts

export class Logger {

  private verbose: boolean;



  constructor(verbose = false) {

    this.verbose = verbose;

  }



  info(message: string) {

    console.log(`[INFO] ${message}`);

  }



  warn(message: string) {

    console.warn(`[WARN] ${message}`);

  }



  error(message: string, error?: Error) {

    console.error(`[ERROR] ${message}`, error || '');

  }



  debug(message: string) {

    if (this.verbose) {

      console.debug(`[DEBUG] ${message}`);

    }

  }



  success(message: string) {

    console.log(`✅ ${message}`);

  }

}
```

Core Logic Skeleton Code

ModuleNode Type Definition (Complete):

JAVASCRIPT

```javascript
// src/graph/module-node.ts

/**
 * Represents a single module in the dependency graph.
 * Follows exact naming convention from requirements.
 */
export class ModuleNode {
  /** Unique identifier (usually the resolved file path) */
  id: string;

  /** Absolute file system path */
  filePath: string;

  /** Modules this module imports (specifier → imported module ID) */
  dependencies: Map<string, string> = new Map();

  /** Modules that import this module */
  dependents: Set<string> = new Set();

  /** Exported identifiers from this module */
  exports: Set<string> = new Set();

  /** Import statements (local name → imported identifier) */
  imports: Map<string, string> = new Map();

  /** Parsed AST for this module */
  ast: object | null = null;

  /** Original source code */
  source: string = '';

  /** Whether this module has side effects */
  sideEffects: boolean = true;

  /** Whether this is an entry point module */
  isEntry: boolean = false;
```

```
  /** Exports that are actually used by other modules (for tree shaking) */
  usedExports: Set<string> = new Set();

  constructor(id: string, filePath: string, isEntry = false) {
    this.id = id;
    this.filePath = filePath;
    this.isEntry = isEntry;
  }

  /**
   * Add a dependency to another module
   * @param specifier Import specifier as written in source (e.g., './utils')
   * @param moduleId Resolved module ID (absolute path)
   */
  addDependency(specifier: string, moduleId: string) {
    this.dependencies.set(specifier, moduleId);
  }

  /**
   * Add an export to this module's public interface
   * @param exportName Name of the exported identifier
   */
  addExport(exportName: string) {
    this.exports.add(exportName);
    // Initially assume all exports are unused until tree shaking
  }

  /**
   * Mark an export as used (called during tree shaking)
   * @param exportName Name of the exported identifier
   */
  markExportUsed(exportName: string) {
    this.usedExports.add(exportName);
  }

  /**
   * Check if this module has any used exports (or has side effects)
   * @returns boolean indicating if module should be included in bundle
   */
```

```
  shouldIncludeInBundle(): boolean {
    // Include if: it's an entry, has side effects, or has used exports
    return this.isEntry || this.sideEffects || this.usedExports.size > 0;
  }
}
```

GraphBuilder Skeleton (TODOs for Learner):

JAVASCRIPT

```javascript
// src/graph/builder.ts

import { ModuleNode } from './module-node.js';

import { Parser } from '../parser/index.js';

import { Resolver } from '../resolver/index.js';

import { Logger } from '../utils/logger.js';


/**

 * Builds the complete module dependency graph from entry points.

 * Orchestrates parsing and resolution to construct the graph.

 */

export class GraphBuilder {

  private parser: Parser;

  private resolver: Resolver;

  private logger: Logger;

  private graph: Map<string, ModuleNode> = new Map();


  constructor(parser: Parser, resolver: Resolver, logger: Logger) {

    this.parser = parser;

    this.resolver = resolver;

    this.logger = logger;

  }


  /**

   * Builds complete module graph from entry points

   * @param entryPaths Array of absolute paths to entry files

   * @returns Promise resolving to the complete ModuleGraph

   */

  async build(entryPaths: string[]): Promise<ModuleGraph> {

    // TODO 1: Validate entryPaths exist and are readable files



    // TODO 2: Create ModuleNode for each entry point and add to graph



    // TODO 3: Process each entry point with BFS/DFS traversal:

    //   - Parse the module to get its dependencies

    //   - For each dependency specifier:

    //       a. Resolve it to absolute path using this.resolver

    //       b. If not already in graph, create ModuleNode and add

    //       c. Add dependency relationship between current module and resolved module
```

```
    // TODO 4: Continue until all dependencies are processed (no new modules)

    // TODO 5: Handle circular dependencies by tracking visited modules in current path

    // TODO 6: Return the completed ModuleGraph instance

    throw new Error('GraphBuilder.build() not implemented');
  }

  /**
   * Process a single module: parse it and resolve its dependencies
   * @param moduleId ID of module to process
   */
  private async processModule(moduleId: string): Promise<void> {
    // TODO 1: Get ModuleNode from graph for moduleId

    // TODO 2: Parse module using this.parser.parseFile(moduleNode.filePath)
    //   - Store AST in moduleNode.ast
    //   - Extract imports/dependencies from AST
    //   - Extract exports from AST and add to moduleNode.exports

    // TODO 3: For each import specifier found:
    //   - Resolve using this.resolver.resolve(specifier, dir of current module)
    //   - If resolution fails, throw descriptive error
    //   - Add dependency relationship

    // TODO 4: Check for package.json sideEffects field and set moduleNode.sideEffects

    // TODO 5: Handle dynamic imports (import()) as potential code split points
    //   Note: For single-bundle output, we still need to include these modules
  }
}
```

Language-Specific Hints

1. **Use ES Modules throughout**: Configure `package.json` with `"type": "module"` to use ES modules in Node.js. This aligns with the modern JavaScript ecosystem we're building for.

2. **TypeScript Configuration**: Use `tsc` with `--module nodenext` and `--moduleResolution nodenext` to accurately mimic Node's ESM resolution behavior during development.
3. **Async File Operations**: Use `fs.promises` API for all file system operations to avoid blocking the event loop during bundling.
4. **Path Handling**: Always use `path.resolve()`, `path.join()`, and `path.relative()` instead of string concatenation to ensure cross-platform compatibility.
5. **Error Messages**: Provide detailed error messages that include:
    - The problematic module specifier
    - The file that contains the import
    - The resolution path that was attempted
    - Suggested fixes when possible

Milestone Checkpoint

After implementing the goals for each milestone, verify functionality with these commands:

Milestone 1 Checkpoint (Parsing):

```bash
# Test parsing a simple module with imports
node src/cli/index.js parse --entry ./test/fixtures/simple-app/index.js
```

Expected: Output showing parsed dependencies: `{ "./utils": "/absolute/path/to/utils.js", "lodash": "node_modules/lodash/lodash.js" }`

Milestone 2 Checkpoint (Resolution):

```bash
# Test resolving a bare specifier
node src/cli/index.js resolve --specifier lodash --from ./test/fixtures/simple-app
```

Expected: Absolute path to lodash in node_modules, e.g., `/project/node_modules/lodash/lodash.js`

Milestone 3 Checkpoint (Bundle Generation):

```bash
# Generate a complete bundle
node src/cli/index.js bundle --entry ./test/fixtures/simple-app/index.js --output ./dist/bundle.js
```

Expected: A working bundle file at `./dist/bundle.js` that can be executed in Node.js with `node dist/bundle.js`

Milestone 4 Checkpoint (Tree Shaking):

```bash
# Bundle with tree shaking enabled
node src/cli/index.js bundle --entry ./test/fixtures/tree-shake-app/index.js --output ./dist/shaken.js --treeshake
```

Expected: Bundle size reduction compared to non-shaken bundle, with console output showing removed exports.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Module not found" error	Incorrect resolution algorithm	Log the resolution path attempted for the specifier	Check `package.json` field priority order
Runtime error: `x is not defined`	Incorrect import/export rewriting	Inspect generated bundle for that module's wrapper	Verify export name mapping in runtime
Bundle includes unused code	Tree shaking not working	Check if `usedExports` set is being populated	Ensure export usage is traced from entry points
Source maps don't work	Incorrect position mapping	Use `source-map` library's `SourceMapConsumer` to test	Verify line/column offsets in generator
Circular dependency hang	Infinite recursion in graph building	Add visited set to `processModule`	Implement cycle detection with stack tracking

High-Level Architecture

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section presents the **end-to-end architecture** of our JavaScript bundler. Think of the bundler as a specialized factory assembly line where raw JavaScript modules enter at one end, undergo systematic transformation and optimization, and emerge as polished, consolidated bundles at the other. The architecture follows a **pipeline pattern** where each component specializes in one aspect of the bundling process, passing its output to the next stage. This clear separation of concerns makes the system testable, understandable, and extensible.

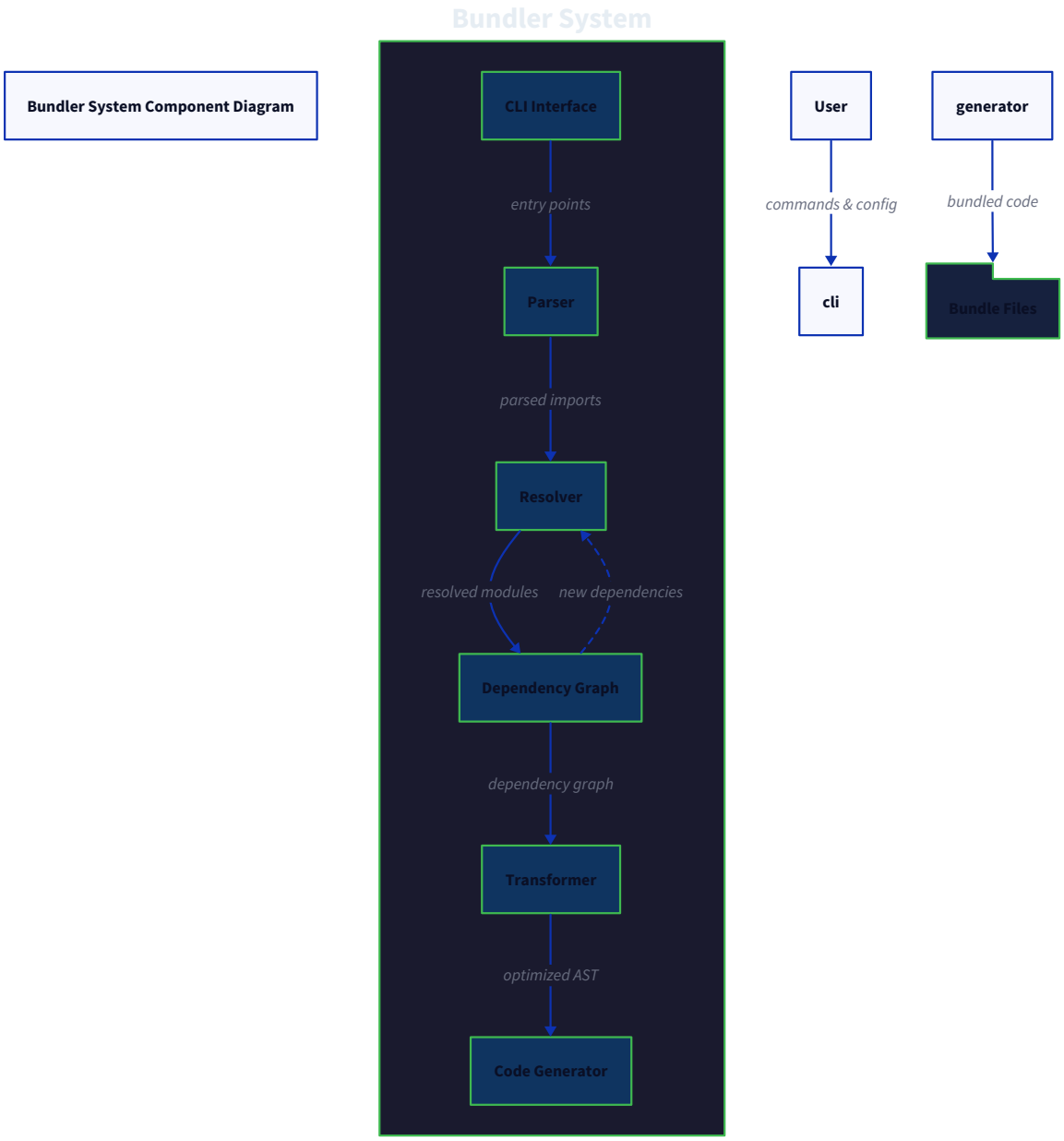
System Overview

Mental Model: The Book Consolidation Factory
Imagine you're running a publishing factory tasked with consolidating thousands of scattered book chapters (JavaScript modules) into organized volumes (bundles). Each chapter references other chapters it depends on (imports). Your assembly line has five specialized workstations:

1. **Chapter Analyzer (Parser)** – Examines each chapter to identify all external references
2. **Address Lookup Service (Resolver)** – Translates chapter references into precise shelf locations
3. **Dependency Mapper (Graph Builder)** – Creates a complete map showing how all chapters interconnect
4. **Content Optimizer (Transformer)** – Removes unused paragraphs and streamlines the content
5. **Volume Assembler (Code Generator)** – Binds the essential chapters into final volumes with a reading guide

This mental model captures the essence of bundling: analyzing dependencies, resolving locations, understanding relationships, optimizing content, and finally packaging everything together with runtime coordination code.

The bundler's core pipeline transforms entry point modules into optimized bundles through five sequential components, each with distinct responsibilities:



Component Flow and Data Transformation:

1. **Parser & Dependency Extractor** – Accepts file paths, reads source code, produces **Abstract Syntax Trees (ASTs)** and extracts all module dependencies (import/export statements).
2. **Module Resolver** – Accepts unresolved module specifiers (like `./utils` or `lodash`) and the current file's directory, returns **absolute file system paths** to concrete module files.
3. **Graph Builder & Transformer** – Accepts entry point paths, orchestrates parsing and resolution to build a complete `ModuleGraph`, then performs **tree shaking** to eliminate unused code.
4. **Code Generator & Runtime** – Accepts the optimized `ModuleGraph`, generates wrapped module code and runtime loader, produces final `BundleChunk` **(s)** with source maps.
5. **CLI/API Interface** – The user-facing layer that accepts configuration, orchestrates the pipeline, and writes output files to disk.

Each component transforms data in a specific format, with the `ModuleGraph` serving as the central data structure that flows through the latter stages of the pipeline. The following table details each component's responsibilities, inputs, and outputs:

Component	Primary Responsibility	Inputs	Outputs	Key Data Structures
Parser & Dependency Extractor	Converts source code into structured AST and extracts all import/export statements	- File path - Source code string - File extension	- AST object - List of unresolved dependencies with specifiers - List of exported identifiers	`ModuleNode.ast` `ModuleNode.imports` `ModuleNode.exports`
Module Resolver	Translates module specifiers to absolute file system paths following Node.js rules	- Unresolved specifier (e.g., `./utils`) - Base directory for resolution	- Absolute file path - Resolved module ID - Package.json metadata (if applicable)	Resolved dependency mapping in `ModuleNode.dependencies`
Graph Builder & Transformer	Constructs complete dependency graph and performs tree shaking optimization	- Entry point paths - Parser and Resolver instances	- Complete `ModuleGraph` with all transitive dependencies - Optimized graph with marked used exports	`ModuleGraph` object `ModuleNode.usedExports` `ModuleNode.sideEffects`
Code Generator & Runtime	Generates executable bundle code with module isolation and loading runtime	- Optimized `ModuleGraph` - Source map options - Output format configuration	- Bundle code string - Source map object - Optional multiple chunks	`BundleChunk` objects
CLI/API Interface	User interaction, configuration management, and pipeline orchestration	- Command-line arguments - Configuration file - Entry point specification	- Written bundle files to disk - Console output and errors	Configuration object Build result metadata

Critical Architectural Insight: The Unidirectional Data Flow

> The pipeline follows a strict **unidirectional data flow**: each component receives only the data it needs from the previous stage and cannot modify earlier stages' internal state. This design eliminates subtle bugs caused by mutable shared state and makes the system easier to reason about. For example, the Graph Builder depends on the Parser and Resolver but doesn't modify their parsing or resolution logic—it only consumes their outputs.

Pipeline Execution Sequence:

1. The CLI receives entry point paths and configuration
2. The Graph Builder initiates the process by requesting the Parser to parse the first entry file
3. For each dependency found, the Resolver converts specifiers to absolute paths
4. The Graph Builder recursively processes new modules until the entire dependency graph is built
5. The Transformer performs tree shaking by marking used exports starting from entry points
6. The Code Generator traverses the optimized graph in topological order, generating wrapped module code
7. The runtime loader template is combined with module code to produce the final bundle
8. Source maps are generated by tracking position mappings through all transformations
9. The bundle is written to the output file system location

This architecture supports incremental extension points: new module formats can be added by extending the Parser, custom resolution logic via the Resolver, additional optimizations in the Transformer, and different output formats in the Code Generator.

Recommended File Structure

A well-organized codebase mirrors the architectural separation of concerns and makes the system more maintainable. The following directory structure groups related functionality together while providing clear boundaries between components:

```
bundler-project/
├── package.json
├── tsconfig.json                  # TypeScript configuration (if using TypeScript)
├── bin/
│   └── bundler                    # CLI entry point (shebang script)
├── src/
│   ├── index.ts                   # Main API entry point
│   ├── cli/
│   │   ├── index.ts               # CLI argument parsing
│   │   ├── config.ts              # Configuration loading and validation
│   │   └── logger.ts              # Colored output, progress indicators
│   ├── parser/
│   │   ├── index.ts               # Parser facade interface
│   │   ├── javascript-parser.ts   # JS/TS AST parsing with Babel/Acorn
│   │   ├── dependency-extractor.ts  # AST traversal for imports/exports
│   │   └── types.ts               # Parser-specific type definitions
│   ├── resolver/
│   │   ├── index.ts               # Resolver facade interface
│   │   ├── node-resolver.ts       # Node.js resolution algorithm
│   │   ├── package-json.ts        # package.json field interpretation
│   │   └── cache.ts               # Resolution caching (optional)
│   ├── graph/
│   │   ├── index.ts               # Graph Builder main interface
│   │   ├── module-node.ts         # ModuleNode class implementation
│   │   ├── module-graph.ts        # ModuleGraph class implementation
│   │   ├── tree-shaker.ts         # Tree shaking algorithm
│   │   └── topological-sort.ts    # Dependency ordering algorithm
│   ├── generator/
│   │   ├── index.ts               # Code Generator facade
│   │   ├── runtime.ts             # Runtime loader template
│   │   ├── module-wrapper.ts      # Module wrapping logic
│   │   ├── source-map.ts          # Source map generation
│   │   └── chunk-splitter.ts      # Code splitting logic (optional)
│   └── utils/
│       ├── fs-cache.ts            # File system caching layer
│       ├── errors.ts              # Custom error types
│       └── async-queue.ts         # Concurrent processing utilities
├── test/
│   ├── fixtures/                  # Test projects for integration tests
│   │   ├── simple-app/
│   │   ├── node-modules/
│   │   └── circular-deps/
│   ├── unit/
│   │   ├── parser.test.ts
│   │   ├── resolver.test.ts
│   │   └── graph.test.ts
│   └── integration/
│       └── bundler.test.ts
└── examples/                      # Example projects for documentation
    ├── simple-bundle/
    ├── code-splitting/
    └── tree-shaking-demo/
```

Directory Rationale:

- **`src/cli/`** – Contains all command-line interface logic, separate from core bundling logic to allow using the bundler programmatically.
- **`src/parser/`** – Isolates AST manipulation and dependency extraction, making it easy to swap parsers or add support for new syntax.
- **`src/resolver/`** – Encapsulates the complex Node.js resolution algorithm, which can be tested independently of file I/O with mock filesystems.
- **`src/graph/`** – Houses the core data structures (`ModuleNode` , `ModuleGraph`) and the tree shaking algorithms that operate on them.
- **`src/generator/`** – Separates code generation concerns from optimization logic, allowing different output formats (IIFE, ESM, CJS) to be implemented as strategies.
- **`src/utils/`** – Contains cross-cutting utilities that don't belong to a specific component, following the "don't repeat yourself" principle.
- **`test/fixtures/`** – Provides complete mini-projects for integration testing, ensuring the bundler works on realistic codebases.
- **`examples/`** – Offers working examples for documentation and user experimentation.

Key Design Decision: Component Independence Through Interfaces

> Each component exposes a clean interface (in its `index.ts`) that other components interact with, rather than importing internal implementation files directly. This allows us to mock components during testing and potentially swap implementations. For example, the Graph Builder depends on `Parser` and `Resolver` interfaces, not concrete `javascript-parser.ts` or `node-resolver.ts`.

Module Boundary Enforcement:

- Components in `parser/` should not import from `generator/`
- The `graph/` component is the only one that directly uses both `parser/` and `resolver/`
- The `cli/` component orchestrates the entire pipeline but doesn't contain bundling logic
- Shared types used across components live in root-level type definitions or are re-exported through interfaces

This structure scales naturally when adding features like plugins (add `src/plugins/`), watch mode (add `src/watcher/`), or custom module systems (add `src/parser/css-module-parser.ts`).

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
AST Parsing	`@babel/parser` with TypeScript plugin	`@babel/parser` with full plugin suite or `swc` for speed
AST Traversal	Custom recursive visitor	`@babel/traverse` for robustness
Path Resolution	Custom implementation of Node algorithm	`resolve` npm package + caching
Source Maps	Basic `source-map` library usage	`magic-string` with source map chains
Concurrency	Sequential processing	Worker threads for parallel parsing
Caching	Memory cache per build	Disk cache with content hashing
File Watching	Not implemented	`chokidar` for filesystem watching

B. Recommended File/Module Structure Implementation

Create the following skeleton files to establish the architecture. Start with the main entry point that ties everything together:

`src/index.ts` – Main API:

TYPESCRIPT

```typescript
import { GraphBuilder } from './graph';
import { Parser } from './parser';
import { Resolver } from './resolver';
import { CodeGenerator } from './generator';
import { FSCache } from './utils/fs-cache';

export interface BundlerOptions {
  entryPoints: string[];
  outDir: string;
  outFile?: string;
  sourceMap?: boolean;
  minify?: boolean;
}

export class Bundler {
  private fsCache: FSCache;
  private parser: Parser;
  private resolver: Resolver;
  private graphBuilder: GraphBuilder;
  private codeGenerator: CodeGenerator;

  constructor(options: BundlerOptions) {
    this.fsCache = new FSCache({ ttl: 60000 });
    this.parser = new Parser({ fsCache: this.fsCache });
    this.resolver = new Resolver({ fsCache: this.fsCache });
    this.graphBuilder = new GraphBuilder({
      parser: this.parser,
      resolver: this.resolver
    });
    this.codeGenerator = new CodeGenerator();
  }

  async build(): Promise<BundleChunk[]> {
    // TODO 1: Call graphBuilder.build() with entryPoints to get ModuleGraph
    // TODO 2: Pass ModuleGraph to codeGenerator.generate() to get BundleChunks
    // TODO 3: Write each BundleChunk to disk in outDir
    // TODO 4: Return array of generated chunks for programmatic usage
  }
```

```
}
```

`src/cli/index.ts` – **CLI Wrapper:**

```typescript
#!/usr/bin/env node

import { Bundler, BundlerOptions } from '../index';
import { loadConfig } from './config';
import { logger } from './logger';

export async function runCLI(args: string[]): Promise<void> {
  try {
    // TODO 1: Parse command line arguments (entry, output, config file)
    // TODO 2: Load configuration from file if specified
    // TODO 3: Merge CLI args with config file options
    // TODO 4: Instantiate Bundler with merged options
    // TODO 5: Call bundler.build() and report progress/success
  } catch (error) {
    logger.error(`Build failed: ${error.message}`);
    process.exit(1);
  }
}

// If this file is being run directly
if (require.main === module) {
  runCLI(process.argv.slice(2));
}
```

C. Infrastructure Starter Code

`src/utils/fs-cache.ts` – **File System Cache (COMPLETE):**

```typescript
import fs from 'fs/promises';

import path from 'path';

import { fileURLToPath } from 'url';


export interface FSCacheOptions {

  ttl: number; // Time-to-live in milliseconds

}


export interface CacheEntry {

  content: string;

  timestamp: number;

  filePath: string;

}


export class FSCache {

  private cache: Map<string, CacheEntry>;

  private ttl: number;


  constructor(options: FSCacheOptions) {

    this.cache = new Map();

    this.ttl = options.ttl;

  }


  async readFile(filePath: string): Promise<string> {

    const normalizedPath = path.resolve(filePath);

    const now = Date.now();



    // Check cache first

    const cached = this.cache.get(normalizedPath);

    if (cached && (now - cached.timestamp) < this.ttl) {

      return cached.content;

    }



    // Read from disk

    try {

      const content = await fs.readFile(normalizedPath, 'utf-8');

      const entry: CacheEntry = {

        content,

        timestamp: now,
```

```
        filePath: normalizedPath
      };

      this.cache.set(normalizedPath, entry);
      return content;
    } catch (error) {
      throw new Error(`Failed to read file ${normalizedPath}: ${error.message}`);
    }
  }

  async fileExists(filePath: string): Promise<boolean> {
    try {
      await fs.access(filePath);
      return true;
    } catch {
      return false;
    }
  }

  async stat(filePath: string): Promise<fs.Stats> {
    return fs.stat(filePath);
  }

  invalidate(filePath: string): void {
    const normalizedPath = path.resolve(filePath);
    this.cache.delete(normalizedPath);
  }

  clear(): void {
    this.cache.clear();
  }
}
```

D. Core Logic Skeleton Code

`src/graph/module-graph.ts` – ModuleGraph Class (SKELETON):

TYPESCRIPT

```typescript
import { ModuleNode } from './module-node';

export class ModuleGraph {
  public nodes: Map<string, ModuleNode>;

  constructor() {
    this.nodes = new Map();
  }

  addModule(module: ModuleNode): void {
    // TODO 1: Check if module with same ID already exists
    // TODO 2: Add module to nodes map using its ID as key
    // TODO 3: If module has dependencies, ensure they're tracked in the graph
  }

  getModule(id: string): ModuleNode | undefined {
    // TODO: Return module by ID or undefined if not found
  }

  hasModule(id: string): boolean {
    // TODO: Check if module exists in graph
  }

  build(entryPaths: string[]): Promise<ModuleGraph> {
    // TODO 1: Create promise-based implementation
    // TODO 2: For each entry path, parse and resolve to get initial ModuleNode
    // TODO 3: Recursively process dependencies of each module
    // TODO 4: Handle circular dependencies by checking visited set
    // TODO 5: Return populated ModuleGraph
  }

  markUsedExports(entryModuleId: string): void {
    // TODO 1: Start with entry module, mark all its exports as used
    // TODO 2: For each used export, trace through import relationships
    // TODO 3: Recursively mark exports used by other modules
    // TODO 4: Handle re-exports (export { x } from './module')
    // TODO 5: Stop when no new exports are marked
  }

  shake(): ModuleGraph {
```

```
    // TODO 1: Create a copy of the graph or mark nodes for removal
    // TODO 2: Remove modules with no used exports and no side effects
    // TODO 3: Remove unused exports from remaining modules
    // TODO 4: Update dependency relationships between remaining modules
    // TODO 5: Return optimized graph
  }

  topologicalSort(): ModuleNode[] {
    // TODO 1: Implement Kahn's algorithm or DFS topological sort
    // TODO 2: Handle circular dependencies by detecting cycles
    // TODO 3: Return array of modules in correct execution order
  }
}
```

E. Language-Specific Hints (JavaScript/TypeScript)

1. **Use ES Modules throughout**: Configure `package.json` with `"type": "module"` and use `.js` / `.ts` extensions in imports for consistency.
2. **TypeScript configuration**: Use `strict: true` in `tsconfig.json` to catch common errors early. Define clear interfaces for cross-component communication.
3. **AST parsing**: When using `@babel/parser`, enable the `sourceType: 'module'` option and plugins for TypeScript, JSX, and latest ECMAScript features.
4. **Async processing**: Use `Promise.allSettled()` for concurrent file reads and parsing, but be mindful of filesystem limits.
5. **Error handling**: Create custom error classes (e.g., `ModuleNotFoundError`, `ParseError`) with helpful metadata for debugging.
6. **Path manipulation**: Always use `path.resolve()` and `path.join()` instead of string concatenation for cross-platform compatibility.
7. **Source maps**: Use the `source-map` library's `SourceMapGenerator` and track positions through each transformation stage.

F. Milestone Checkpoint

After implementing the basic architecture skeleton, verify your setup with:

```bash
# 1. Check TypeScript compilation
npx tsc --noEmit

# 2. Run the CLI with a simple test file
mkdir -p test-project
echo "export const hello = 'world';" > test-project/index.js
node ./bin/bundler --entry ./test-project/index.js --outdir ./dist

# Expected behavior:
# - No TypeScript errors in the architecture files
# - The CLI should run without crashing (even if it doesn't produce output yet)
# - You should see a basic log message or error about missing implementations

# 3. Verify module resolution
npm test -- --testPathPattern=utils/fs-cache

# Expected: File cache tests should pass, demonstrating the infrastructure layer works
```

Signs of correct architecture:

- You can import `FSCache` in parser tests without circular dependencies
- The `ModuleGraph` class compiles without implementation but with correct method signatures
- Running the CLI shows a help message or recognizes basic arguments

Common early issues to check:

- **Module resolution errors**: Ensure `package.json` has `"type": "module"` if using ES modules
- **TypeScript path aliases**: Configure `tsconfig.json` `baseUrl` and `paths` if using absolute imports
- **Missing dependencies**: Install `@babel/parser`, `@types/node`, and `source-map` as devDependencies

Data Model

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

The **data model** forms the structural foundation of our bundler, defining how we represent code modules, their relationships, and the eventual output. Think of this as the architectural blueprint for the entire system—if the bundler is a factory transforming raw materials into finished products, the data model specifies the standardized containers and assembly lines that hold everything together during processing. These data structures persist throughout the bundling pipeline, evolving as we progress from parsing source files to generating optimized bundles.

A well-designed data model provides three critical benefits: **traceability** (we can always trace a piece of bundled code back to its original module), **efficiency** (we can perform tree shaking and ordering without redundant calculations), and **correctness** (we preserve the precise import/export relationships that JavaScript modules require at runtime). The following three core types— `ModuleNode`, `ModuleGraph`, and `BundleChunk` —work together to achieve these goals.

Module Node

Mental Model: The DNA Blueprint Think of a `ModuleNode` as the complete genetic blueprint for a single JavaScript file. Just as DNA encodes all the information needed to build and operate an organism—its structure (genes), dependencies on other molecules (imports), and capabilities it provides to the system (exports)—the `ModuleNode` encodes everything about a source module. It's not the living code itself but the structured metadata that allows us to analyze, transform, and ultimately reproduce it correctly in the bundled output. Each blueprint is unique to its file, but references other blueprints in a network of dependencies.

A `ModuleNode` is the atomic unit of our bundling system, representing a single source file after it has been parsed and analyzed. It contains both **static metadata** (file location, unique identifier) and **dynamic analysis results** (parsed AST, extracted imports/exports, side-effect flags). This combination allows us to make intelligent bundling decisions without repeatedly re-parsing files.

Field	Type	Description
`id`	`string`	Unique identifier for the module, typically a normalized absolute path or a generated hash. Used as the primary key in the module graph and as the reference in the runtime module registry.
`filePath`	`string`	Absolute filesystem path to the source file (e.g., `/project/src/utils.js`). Used for resolution, source map generation, and debug output.
`dependencies`	`Map<string, string>`	Mapping from import specifiers (as they appear in source code, like `"./helper"` or `"lodash"`) to resolved module IDs. Keys are the raw specifiers; values are the `id` of the target `ModuleNode`.
`dependents`	`Set<string>`	Inverse of dependencies: set of module IDs that import *this* module. Crucial for tree shaking to know which modules depend on a given export.
`exports`	`Set<string>`	All named exports this module provides (e.g., `["sum", "average"]`). Includes `"default"` if the module has a default export. Used for export tracking during tree shaking.
`imports`	`Map<string, string>`	Mapping from local import names to the exported names they reference (e.g., `{ localName: "sum", imported: "calculateSum" }` represented as `"sum" → "calculateSum"`). Enables precise rewriting of import references during code generation.
`ast`	`object` (ESTree)	The parsed Abstract Syntax Tree of the module's source code. Allows static analysis without re-parsing and serves as input for transformations like import/export rewriting.
`source`	`string`	Original source code of the module. Preserved for source map generation and debugging.
`sideEffects`	`boolean`	Whether the module has side effects when evaluated (e.g., polyfills that modify globals, CSS-in-JS registration). Derived from `package.json` `sideEffects` field or static analysis. Determines if an unused module can be safely removed.
`isEntry`	`boolean`	True if this module was specified as an entry point in the bundler configuration. Entry modules are the roots of the dependency graph—their exports become accessible from the bundle's public interface.
`usedExports`	`Set<string>`	Subset of `exports` that are actually imported by other modules (or the entry point's public interface). Initially empty; populated during the tree shaking mark phase. Determines which exports survive to the final bundle.

The `ModuleNode` evolves through the bundling pipeline:

1. **Initial creation** during parsing: `filePath`, `source`, `ast`, `dependencies` (specifiers only) are populated.
2. **Resolution phase**: `dependencies` gets resolved module IDs; `exports` and `imports` are extracted from the AST.
3. **Graph building**: `dependents` is populated as we establish bidirectional links between nodes.
4. **Tree shaking**: `usedExports` is marked based on actual usage; `sideEffects` may be read from `package.json`.
5. **Code generation**: The node's `ast` and `usedExports` guide the transformation and wrapping process.

> **Design Insight:** We store both `dependencies` (outgoing) and `dependents` (incoming) because tree shaking fundamentally works by traversing *from dependents to dependencies*—we need to know which modules import a given export to determine if it's used. This bidirectional linking turns the graph into an undirected structure for traversal purposes while maintaining the directed semantics for execution order.

Common Operations on ModuleNode

Method	Parameters	Returns	Description
`addDependency(specifier, moduleId)`	`specifier: string`, `moduleId: string`	`void`	Records that this module imports `specifier`, which resolves to the module with `moduleId`. Updates the `dependencies` map.
`addExport(exportName)`	`exportName: string`	`void`	Adds an export name to the `exports` set. Called during AST analysis for each `export` statement found.
`markExportUsed(exportName)`	`exportName: string`	`void`	Adds the export to the `usedExports` set, marking it as live for tree shaking. Called during the mark phase.
`shouldIncludeInBundle()`	None	`boolean`	Returns `true` if the module should be included in the final bundle. A module is included if: (1) it's an entry point (`isEntry === true`), OR (2) any of its exports are in `usedExports`, OR (3) `sideEffects === true` (indicating it must run even if exports are unused).

Example Walkthrough: Analyzing a Utility Module Consider a file `/src/math.js`:

```javascript
import { format } from './formatter';

export const sum = (a, b) => format(a + b);

export const average = (arr) => sum(...arr) / arr.length;

export default sum;
```

After parsing and analysis, its `ModuleNode` would contain:

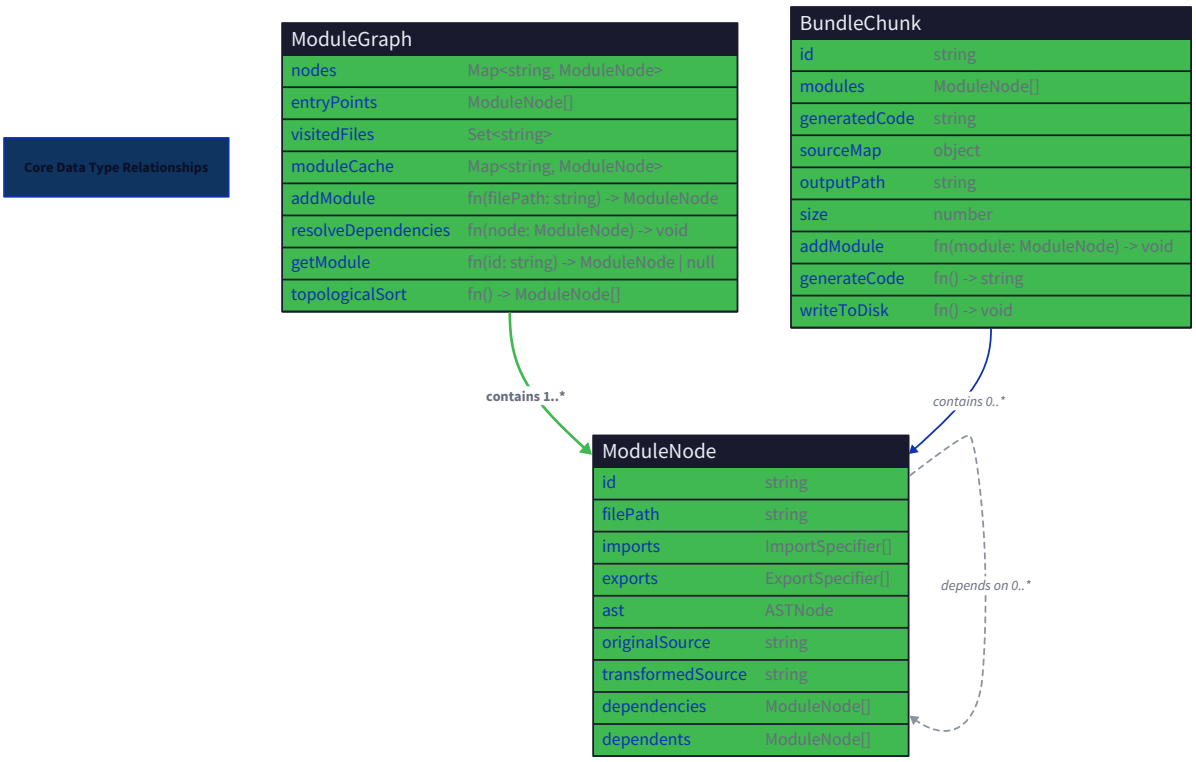
- `id`: `/src/math.js` (normalized absolute path)
- `exports`: `Set { "sum", "average", "default" }`
- `dependencies`: `Map { "./formatter" → "/src/formatter.js" }`
- `imports`: `Map { "format" → "format" }` (local name → imported name)
- `sideEffects`: `false` (assuming no top-level side effects)
- `usedExports`: Initially empty; later populated if other modules import `sum` or `average`

Module Graph

Mental Model: The Transportation Network Map Imagine the `ModuleGraph` as a detailed map of a city's transportation system. Each `ModuleNode` is a station, and the import relationships are train lines connecting them. Some stations are major hubs (entry points), while others are smaller stops (utility modules). The map shows not just the routes (dependencies) but also passenger flow (export usage)—which stations are actually visited during a typical journey. This map allows us to optimize the network by removing unused routes and stations, ensuring the final bundled output contains only the necessary infrastructure for the application to run.

The `ModuleGraph` is a **directed graph** (edges go from importer to dependency) with bidirectional navigation support (via `dependents`). It serves as the central data structure that holds the complete dependency analysis of the entire application, enabling global optimizations like tree shaking and topological sorting.

Field	Type	Description
`nodes`	`Map<string, ModuleNode>`	Primary storage mapping module IDs to their corresponding `ModuleNode` objects. Provides O(1) lookup by ID and ensures each module appears only once in the graph.



Decision: Bidirectional Edges via Explicit Dependent Tracking

- **Context**: We need efficient reverse traversal for tree shaking (finding all importers of a given export) and for determining if a module is orphaned.
- **Options Considered**:
  1. **Unidirectional adjacency list**: Store only `dependencies` in each node, compute reverse edges on demand via graph scan.
  2. **Bidirectional adjacency list**: Store both `dependencies` and `dependents` explicitly in each node.
  3. **Separate reverse index**: Maintain a global `Map<string, Set<string>>` of dependents separate from nodes.
- **Decision**: Store `dependents` as a `Set` within each `ModuleNode` (option 2).
- **Rationale**:
  - **Traversal efficiency**: Tree shaking requires frequent "who imports this?" queries. With explicit dependents, this is O(1) per module vs O(n) graph scan.
  - **Data locality**: A module's dependents are intrinsic to that module—keeping them together improves cache locality and simplifies serialization/debugging.
  - **Implementation simplicity**: No separate data structure to keep synchronized; the graph builder naturally populates both directions.
- **Consequences**:
  - **Memory overhead**: Each module stores an extra `Set`, but typical module graphs have limited fan-in (few importers per module).
  - **Maintenance complexity**: Graph modifications must update both sides of the relationship, but this is encapsulated in `ModuleGraph` methods.

Option	Pros	Cons	Chosen?
Unidirectional	Minimal memory, simple updates	Reverse queries require O(n) scan	No
Bidirectional (in-node)	O(1) reverse queries, data locality	Extra memory for `Set`	**Yes**
Separate reverse index	Clean separation, efficient queries	Synchronization complexity, extra abstraction	No

Key Graph Operations

Method	Parameters	Returns	Description
`addModule(module)`	`module: ModuleNode`	`void`	Inserts a new module into the graph. Ensures the module's ID doesn't already exist.
`build(entryPaths)`	`entryPaths: string[]`	`Promise<ModuleGraph>`	High-level method that orchestrates parsing and resolution starting from entry paths, returning the fully built graph. Internally calls parser, resolver, and connects nodes.
`markUsedExports(entryModuleId)`	`entryModuleId: string`	`void`	Initiates tree shaking by marking all exports reachable from the given entry module. Performs a graph traversal, calling `markExportUsed` on visited nodes.
`shake()`	None	`ModuleGraph`	Returns a new graph (or mutates this one) with unused modules and exports removed. Prunes nodes where `shouldIncludeInBundle()` returns `false`.
`topologicalSort()`	None	`ModuleNode[]`	Returns modules in **post-order dependency order**: dependencies before dependents. Ensures runtime execution order satisfies import relationships.

Algorithm: Building the Module Graph

1. **Initialize**: Create empty `ModuleGraph` and a queue of modules to process.
2. **Start from entries**: For each entry path in configuration:
   - Resolve to absolute path (handling `package.json` fields).
   - Create initial `ModuleNode` with `isEntry: true`.
   - Add to graph and enqueue for processing.
3. **Process queue** (BFS/DFS):
   - Dequeue next module.
   - Parse its source to AST, extract raw import specifiers.
   - For each specifier:
     - Resolve to absolute path using Node.js algorithm.
     - If resolved module not already in graph:
       - Create new `ModuleNode` (non-entry).
       - Add to graph and enqueue.
     - Link the dependency:
       - Call `importer.addDependency(specifier, resolvedId)`.
       - Add importer's ID to dependency's `dependents` set.
4. **Repeat** until queue empty, building the transitive closure of all reachable modules.

Algorithm: Tree Shaking via Mark-and-Sweep

1. **Mark phase** (starting from entries):
   - For each entry module: mark all its exports as used (since entry exports are publicly accessible).
   - For each used export in a module:
     - Find all importing modules via `dependents`.
     - For each importer: mark the specific imported name as used in that importer's context.
     - Recursively process the imported module's used exports.
2. **Sweep phase**:
   - Iterate through all modules in graph.
   - Remove any module where `shouldIncludeInBundle()` returns `false`.

- For remaining modules: filter their `exports` list to only those in `usedExports`.

Example: Graph with Re-exports Consider three files:

- `a.js`: `export const secret = 1; export const public = 2;`
- `b.js`: `export { public } from './a';`
- `entry.js`: `import { public } from './b'; console.log(public);`

The graph will show: `entry` → `b` → `a`. During tree shaking:

- `entry` uses `public` from `b` → marks `b.public` as used.
- `b.public` re-exports `a.public` → marks `a.public` as used.
- `a.secret` remains unmarked → excluded from bundle.
- Result: Only `a.public` and its dependent code included.

Bundle Chunk

Mental Model: The Shipping Container A `BundleChunk` is like a standardized shipping container prepared for deployment. It contains a carefully selected set of modules (like cargo items), all packaged together with protective wrapping (runtime code) and a manifest (source map) that describes exactly where each item came from and how it was arranged. Just as shipping containers optimize cargo handling by grouping items with the same destination, chunks group modules that should be loaded together—either immediately (main chunk) or on-demand (split chunks).

While our initial implementation focuses on a single bundle, the `BundleChunk` abstraction prepares for **code splitting**, where multiple output files are generated for different parts of the application. Each chunk is an independent JavaScript file that can be loaded separately by the browser.

Field	Type	Description
`id`	`string`	Identifier for the chunk, often derived from entry point name (e.g., `"main"`) or dynamic import pattern. Used in naming output files and in runtime loading logic.
`modules`	`Set<string>`	Set of module IDs included in this chunk. Determines which modules' code appears in the final output file.
`code`	`string`	The generated JavaScript source code for this chunk, including: (1) runtime module loader, (2) wrapped module functions, (3) initialization code that executes entry modules.
`sourceMap`	`object` (V3)	Source map following the v3 specification, mapping positions in `code` back to original source files. Enables debugging original source in browser dev tools.

Chunk Formation Strategies

Strategy	When Used	Example
Single entry chunk	Default for simple apps	One chunk containing all modules reachable from main entry.
Multiple entry chunks	Multiple `entryPoints` config	Separate chunks for `app.js` and `admin.js`, with shared modules possibly duplicated or split.
Dynamic import splits	`import()` calls in code	`main.js` (initial load) + `charting.js` (loaded when user views charts).
Vendor splitting	Manual optimization	`vendor-react.js` containing React and its dependencies, separate from app code.

Relationship to Module Graph A `BundleChunk` is essentially a **view** or **subset** of the `ModuleGraph` selected for co-location in a single output file. The selection logic depends on the bundling strategy:

1. **For single-bundle projects**: One chunk containing all modules from the graph.
2. **For code splitting**: Multiple chunks, each with modules selected by:
   - Entry point isolation (each entry gets its own chunk)
   - Dynamic import boundaries (modules imported via `import()` go to separate chunks)
   - Manual configuration (e.g., "put all node_modules in vendor chunk")

> **Design Insight:** We separate `modules` (which modules are included) from `code` (the generated JavaScript) because the same set of modules could theoretically be bundled differently—with different wrapping strategies, minification levels, or runtime implementations. This separation also allows us to compute chunk metadata (like total size) before generating actual code.

Example: Multi-Chunk Scenario Given modules: `entry.js`, `utils.js`, `heavy.js` (dynamically imported), `vendor.js`. The bundler might produce:

- **Chunk A** (`id: "main"`, `modules: {"entry.js", "utils.js", "vendor.js"}`): Loaded initially.
- **Chunk B** (`id: "heavy"`, `modules: {"heavy.js"}`): Loaded on-demand when `import('./heavy')` executes.

Both chunks share the same module runtime pattern but contain different module registrations.

Common Pitfalls

⚠️ **Pitfall: Misunderstanding Module Node Identity The Mistake**: Using the raw import specifier (e.g., `"./utils"`) as the module ID, or creating duplicate nodes for the same physical file when it's reached via different resolution paths (e.g., symlinks, `package.json` `exports` aliases). **Why It's Wrong**: This breaks the fundamental graph property that each file corresponds to exactly one node. Duplicate nodes cause:

- Tree shaking incorrectly removing "unused" exports that are actually used through the other node.
- Code duplication in the bundle (same module included multiple times).
- Runtime errors (different module instances with separate state). **The Fix**: Always **normalize** resolved paths to a canonical form before using as ID. Use `fs.realpath()` to resolve symlinks, and apply consistent slash direction and case normalization. Store the original specifier in `dependencies` mapping, but use canonical path for `id`.

⚠️ **Pitfall: Incomplete Export Tracking for Re-exports The Mistake**: Treating `export { foo } from './bar'` as just a dependency edge without preserving the re-export relationship during analysis. **Why It's Wrong**: Tree shaking needs to know that `foo` in the re-exporting module is actually `bar.foo`. If we only track dependencies, we lose the semantic link needed to mark `bar.foo` as used when someone imports `foo` from the re-exporting module. **The Fix**: During AST analysis, create a special record for re-exports that captures both the dependency (target module) and the specific export name mapping. During tree shaking, follow these re-export chains transitively.

⚠️ **Pitfall: Ignoring Module Execution Side Effects The Mistake**: Assuming a module with no used exports can always be removed from the bundle. **Why It's Wrong**: Many modules have **side effects** when evaluated: polyfills that modify global objects, CSS-in-JS library registration, initialization code that sets up singletons. Removing these breaks the application. **The Fix**:

1. Read `package.json` `"sideEffects": false` field for third-party modules.
2. Perform conservative static analysis: if a module contains top-level function calls (except pure built-ins), assignments to globals, or `new` expressions, treat it as having side effects.
3. Always include modules with `sideEffects: true` even if no exports are used.

⚠️ **Pitfall: Naive Topological Sort with Cycles The Mistake**: Using a standard topological sort algorithm that fails when the graph contains circular dependencies. **Why It's Wrong**: JavaScript modules *do* support circular dependencies through the live bindings mechanism. The runtime must handle initialization of mutually dependent modules. A sort that fails on cycles prevents bundling many real-world codebases. **The Fix**: Use a **post-order DFS** that detects cycles and still produces a valid (if arbitrary) order within cycles. The runtime module loader must handle partially initialized modules—a module's factory function may execute before all its dependencies are fully initialized, which is why we need the "loading" state in the runtime.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
AST Representation	ESTree standard format (used by Babel, Acorn)	Custom lightweight AST with only needed nodes
Graph Storage	JavaScript `Map` and `Set`	Optimized adjacency lists with integer IDs
Source Map Generation	`source-map` library from Mozilla	Custom VLQ encoding for minimal overhead

Recommended File Structure

```
bundler/
├── src/
│   ├── data-model/           # Data structures defined in this section
│   │   ├── ModuleNode.ts     # ModuleNode class with all fields and methods
│   │   ├── ModuleGraph.ts    # ModuleGraph class with graph algorithms
│   │   ├── BundleChunk.ts    # BundleChunk class and chunk splitting logic
│   │   └── index.ts          # Exports all data model types
│   ├── parser/               # Milestone 1: AST parsing
│   ├── resolver/             # Milestone 2: Module resolution
│   ├── transformer/          # Milestone 3 & 4: Tree shaking and code gen
│   └── bundler.ts            # Main pipeline orchestrator
└── tests/
    └── data-model/           # Unit tests for these data structures
```

Complete Starter Code for Core Types

TYPESCRIPT

```typescript
// src/data-model/ModuleNode.ts

import type { Node } from '@babel/types';


/**

 * Represents a single JavaScript/TypeScript module in the dependency graph.

 * Think of it as the "DNA blueprint" for a source file.

 */

export class ModuleNode {

  // Core identity

  public id: string;

  public filePath: string;



  // Graph relationships

  public dependencies: Map<string, string> = new Map();  // specifier → moduleId

  public dependents: Set<string> = new Set();            // modules that import this one



  // Export/import analysis

  public exports: Set<string> = new Set();

  public imports: Map<string, string> = new Map();       // localName → importedName

  public usedExports: Set<string> = new Set();           // populated during tree shaking



  // Source and AST

  public ast: Node | null = null;

  public source: string;



  // Optimization flags

  public sideEffects: boolean = true;  // Conservative default

  public isEntry: boolean = false;



  constructor(id: string, filePath: string, source: string, isEntry: boolean = false) {

    this.id = id;

    this.filePath = filePath;

    this.source = source;

    this.isEntry = isEntry;

  }



  /**
```

```
 * Records a dependency from this module to another.
 * @param specifier - The import specifier as written in source (e.g., "./utils")
 * @param moduleId - The resolved module ID (e.g., "/src/utils.js")
 */
addDependency(specifier: string, moduleId: string): void {
  this.dependencies.set(specifier, moduleId);
}

/**
 * Adds an export name to this module's public interface.
 * @param exportName - Name of the export (use "default" for default exports)
 */
addExport(exportName: string): void {
  this.exports.add(exportName);
}

/**
 * Marks an export as used during tree shaking analysis.
 * @param exportName - Name of the export that is actually imported somewhere
 */
markExportUsed(exportName: string): void {
  this.usedExports.add(exportName);
}

/**
 * Determines if this module should be included in the final bundle.
 * A module is included if:
 * 1. It's an entry point, OR
 * 2. Any of its exports are used, OR
 * 3. It has side effects (even if no exports are used)
 */
shouldIncludeInBundle(): boolean {
  return this.isEntry || this.usedExports.size > 0 || this.sideEffects;
}

/**
```

```
   * Returns a debug-friendly representation of the module.
   */
  toString(): string {
    return `ModuleNode(id=${this.id}, exports=${Array.from(this.exports)}, deps=${Array.from(this.dependencies.keys())})`;
  }
}
```

TYPESCRIPT

```typescript
// src/data-model/ModuleGraph.ts

import { ModuleNode } from './ModuleNode';


/**

 * Directed graph representing the complete module dependency network.

 * Think of it as a "transportation network map" connecting all modules.

 */

export class ModuleGraph {

  // Primary storage: moduleId → ModuleNode

  public nodes: Map<string, ModuleNode> = new Map();



  /**

   * Adds a module to the graph, ensuring no duplicates.

   * @throws Error if a module with the same ID already exists

   */

  addModule(module: ModuleNode): void {

    if (this.nodes.has(module.id)) {

      throw new Error(`Module ${module.id} already exists in graph`);

    }

    this.nodes.set(module.id, module);

  }



  /**

   * Gets a module by ID, returns undefined if not found.

   */

  getModule(id: string): ModuleNode | undefined {

    return this.nodes.get(id);

  }



  /**

   * Creates a bidirectional link between importer and dependency.

   * Call this after resolution to connect nodes.

   */

  linkDependency(importerId: string, specifier: string, dependencyId: string): void {

    const importer = this.nodes.get(importerId);

    const dependency = this.nodes.get(dependencyId);
```

```
    if (!importer || !dependency) {
      throw new Error(`Cannot link non-existent modules: ${importerId} -> ${dependencyId}`);
    }

    // Forward link: importer knows about dependency
    importer.addDependency(specifier, dependencyId);

    // Reverse link: dependency knows who imports it
    dependency.dependents.add(importerId);
  }

  /**
   * High-level graph construction from entry points.
   * TODO: Implement parsing and resolution pipeline
   * Algorithm steps:
   * 1. Initialize queue with entry modules
   * 2. While queue not empty:
   *    a. Dequeue module
   *    b. Parse its AST
   *    c. Extract import specifiers
   *    d. Resolve each specifier to absolute path
   *    e. Create/link new modules as needed
   *    f. Enqueue newly discovered modules
   */
  async build(entryPaths: string[]): Promise<ModuleGraph> {
    // TODO 1: Create ModuleNode for each entry path, mark as entry
    // TODO 2: Add entry nodes to graph and to processing queue
    // TODO 3: While queue not empty, process next module
    // TODO 4: For each module, parse source to AST (use @babel/parser)
    // TODO 5: Extract import/export statements from AST
    // TODO 6: For each import specifier, resolve to absolute path
    // TODO 7: If resolved module not in graph, create and enqueue it
    // TODO 8: Call linkDependency to connect importer to dependency
    // TODO 9: Repeat until queue empty (transitive closure built)
    return this;
  }
```

```
/**
 * Marks all exports reachable from the given entry module.
 * This is the "mark" phase of tree shaking.
 * Algorithm steps:
 * 1. Start with entry module's exports
 * 2. For each used export, find all modules that import it
 * 3. Recursively mark those imports as used
 * 4. Follow re-export chains transitively
 */
markUsedExports(entryModuleId: string): void {
  const entryModule = this.nodes.get(entryModuleId);
  if (!entryModule) return;

  // TODO 1: Initialize worklist with entry module's exports
  // TODO 2: While worklist not empty:
  // TODO 3:   Pop (moduleId, exportName) from worklist
  // TODO 4:   Mark exportName as used in that module
  // TODO 5:   For each dependent module that imports this export:
  // TODO 6:     Add the imported name in dependent module to worklist
  // TODO 7:   For re-exports of this export to other modules, add them to worklist
}

/**
 * Removes unused modules and exports from the graph.
 * Returns a new graph with only live code.
 * Algorithm steps:
 * 1. Run markUsedExports on all entry points
 * 2. Create new ModuleGraph
 * 3. For each module in original graph:
 *    a. If shouldIncludeInBundle() returns true, add to new graph
 *    b. Filter exports to only usedExports
 * 4. Rebuild dependency links in new graph
 */
shake(): ModuleGraph {
  // TODO 1: Run markUsedExports for each entry module in the graph
```

```
    // TODO 2: Create new empty ModuleGraph
    // TODO 3: Iterate through all nodes in this graph
    // TODO 4: For each node, if node.shouldIncludeInBundle() is true:
    // TODO 5:   Create a copy with only usedExports
    // TODO 6:   Add to new graph
    // TODO 7: Re-establish dependency links between nodes in new graph
    // TODO 8: Return the new, shaken graph
    return new ModuleGraph();
  }

  /**
   * Returns modules in topological order (dependencies before dependents).
   * Handles circular dependencies by using post-order DFS.
   * Algorithm steps:
   * 1. Perform DFS, marking visited nodes
   * 2. On finishing a node, push to result array
   * 3. Reverse result to get dependency order
   */
  topologicalSort(): ModuleNode[] {
    const visited = new Set<string>();
    const temp = new Set<string>();  // For cycle detection
    const result: ModuleNode[] = [];

    const visit = (nodeId: string) => {
      if (temp.has(nodeId)) {
        // Cycle detected - this is allowed in JS modules
        return;
      }
      if (visited.has(nodeId)) return;

      const node = this.nodes.get(nodeId);
      if (!node) return;

      temp.add(nodeId);

      // Visit all dependencies first
```

```
      for (const depId of node.dependencies.values()) {
        visit(depId);
      }

      temp.delete(nodeId);
      visited.add(nodeId);
      result.push(node);
    };

    // Start from all nodes (handles disconnected graphs)
    for (const nodeId of this.nodes.keys()) {
      visit(nodeId);
    }

    return result;
  }
}
```

TYPESCRIPT

```typescript
// src/data-model/BundleChunk.ts

/**
 * Represents a single output bundle file.
 * Think of it as a "shipping container" for grouped modules.
 */
export interface SourceMapV3 {
  version: number;
  file?: string;
  sourceRoot?: string;
  sources: string[];
  sourcesContent?: string[];
  names: string[];
  mappings: string;
}

export class BundleChunk {
  public id: string;
  public modules: Set<string>;
  public code: string = '';
  public sourceMap: SourceMapV3 | null = null;

  constructor(id: string, modules: Set<string> = new Set()) {
    this.id = id;
    this.modules = modules;
  }

  /**
   * Adds a module to this chunk.
   */
  addModule(moduleId: string): void {
    this.modules.add(moduleId);
  }

  /**
   * Estimates the size of this chunk in bytes.
   * Useful for code splitting decisions.
   */
```

```
  estimateSize(): number {
    // Rough estimate: assume average 100 bytes per module + runtime overhead
    return this.modules.size * 100 + 1000;
  }

  /**
   * Generates a filename for this chunk.
   * Can be overridden for custom naming strategies.
   */
  getFileName(): string {
    return this.id === 'main' ? 'bundle.js' : `${this.id}.chunk.js`;
  }
}
```

Language-Specific Hints

- **JavaScript/TypeScript**: Use `Map` and `Set` for relationships—they preserve insertion order (helpful for deterministic builds) and provide O(1) lookups.
- **TypeScript**: Define interfaces for AST nodes using `@babel/types` for type safety during AST traversal.
- **Performance**: For large graphs (10k+ modules), consider using integer IDs instead of strings for internal references to reduce memory.
- **Serialization**: Implement `toJSON()` methods on data model classes for debugging output, but avoid serializing the AST (it's huge).

Milestone Checkpoint: Data Model Validation After implementing these data structures, verify they work correctly:

BASH

```
# Run basic unit tests
npm test -- --testPathPattern=data-model

# Or manually test in Node REPL
node -e "
const { ModuleNode, ModuleGraph } = require('./dist/data-model');
const graph = new ModuleGraph();
const entry = new ModuleNode('entry.js', '/src/entry.js', 'import {x} from ./a', true);
graph.addModule(entry);
console.log('Graph has', graph.nodes.size, 'modules');
console.log('Entry module isEntry:', entry.isEntry);
```

Expected Behavior: You should be able to create modules, add them to a graph, link dependencies, and see bidirectional relationships. The `topologicalSort` method should handle simple dependency chains without errors.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Maximum call stack size exceeded" in topological sort	Circular dependency causing infinite recursion in naive DFS	Log visited nodes; check for A→B→C→A patterns	Implement cycle detection with `temp` set as shown
Tree shaking removes obviously used code	`usedExports` not being propagated through re-exports	Debug `markUsedExports` by logging (moduleId, exportName) pairs	Ensure re-exports add both modules to worklist
Module included but all exports empty	`usedExports` set never populated for non-entry modules	Check if `markUsedExports` is called at all	Call `markUsedExports` from all entry points
Duplicate module IDs for same file	Different resolution paths (symlinks, relative vs absolute)	Compare normalized paths using `fs.realpath`	Canonicalize all paths before using as IDs

---

Component 1: Parser & Dependency Extractor

> **Milestone(s):** Milestone 1

The **Parser & Dependency Extractor** is the entry point of our bundler's transformation pipeline. This component takes raw JavaScript/TypeScript source files and transforms them into structured, analyzable data that captures both the code's internal structure and its external dependencies. Think of this as the foundation upon which all subsequent analysis and optimization depends—an incomplete parse will lead to incorrect resolution, broken bundles, and missed optimization opportunities.

Mental Model: The Code Archaeologist

Imagine you're an archaeologist excavating an ancient manuscript. Your goal isn't just to read the text but to understand its structure, note its cross-references to other manuscripts, and document every external citation. The **Parser & Dependency Extractor** performs exactly this role in our bundler ecosystem:

- **Excavation (Parsing)**: Just as an archaeologist carefully brushes away dirt to reveal the manuscript's structure, the parser converts raw source code into an **Abstract Syntax Tree (AST)**—a hierarchical representation where each node corresponds to a language construct (function declarations, variable assignments, import statements).
- **Mapping (AST Construction)**: The archaeologist creates a detailed map of the manuscript's sections, chapters, and paragraphs. Similarly, the AST maps every expression, statement, and declaration in the source file, preserving the original structure and location information.
- **Citation Recording (Dependency Extraction)**: When the archaeologist finds a reference like "see Manuscript B, Chapter 3," they record this cross-reference. The dependency extractor traverses the AST to find all `import`, `export`, and `require` statements, recording the module specifiers (like `"./utils"` or `"lodash"`) that represent dependencies on other modules.
- **Conditional References (Dynamic Import Detection)**: Some manuscripts say "refer to Manuscript C when condition X is true." Similarly, the extractor identifies dynamic `import()` expressions—conditional or deferred module loads that represent potential code-split points.

This mental model emphasizes precision, thoroughness, and structural understanding. Just as an archaeologist's initial documentation affects all subsequent historical analysis, the parser's accuracy determines the correctness of the entire bundling process.

Interface

The Parser & Dependency Extractor exposes a clean, functional interface that hides the complexity of AST manipulation. The primary functions transform source code into structured module data:

Function/Method	Parameters	Returns	Description
`parseModule(sourceCode, filePath)`	`sourceCode: string` - Raw source content `filePath: string` - Absolute path to source file	`{ ast: AST, imports: ImportSpecifier[], exports: ExportSpecifier[], dynamicImports: DynamicImport[] }`	Parses source code into an AST and extracts all static dependency information. Returns structured data about imports, exports, and dynamic imports.
`extractDependencies(ast)`	`ast: AST` - Abstract Syntax Tree from parser	`{ imports: ImportSpecifier[], exports: ExportSpecifier[], dynamicImports: DynamicImport[] }`	Traverses an existing AST to extract dependency information without re-parsing. Used for cached ASTs.
`createModuleNode(filePath, sourceCode, isEntry)`	`filePath: string` - Absolute file path `sourceCode: string` - File contents `isEntry: boolean` - Whether this is an entry point	`ModuleNode` - Initialized module node	Factory function that parses source, extracts dependencies, and creates a populated `ModuleNode` with `dependencies`, `imports`, `exports`, and `ast` fields set.

Data Structures Extracted

The parser returns several structured objects that feed into the module resolution and graph construction phases:

ImportSpecifier (represents a single import statement):

Field	Type	Description
`specifier`	`string`	The raw import string (e.g., `"./utils"`, `"react"`)
`type`	`'static' | 'dynamic'`	Whether this is a static ES import or dynamic `import()`
`imported`	`string | '*' | 'default' | null`	The specific binding being imported (`'Component'`, `'*'` for namespace, `'default'`, `null` for side-effect imports)
`local`	`string`	Local name for the imported binding in current module
`sourceLocation`	`{ line: number, column: number }`	Original location in source file for error reporting

ExportSpecifier (represents a single export):

Field	Type	Description
`exported`	`string`	Name of the exported binding (`'default'` for default exports)
`local`	`string | null`	Local name being exported (`null` for re-exports like `export * from './module'`)
`source`	`string | null`	Module specifier for re-exports (e.g., `"./module"`), `null` for local exports
`type`	`'named' | 'default' | 'namespace' | 'all'`	Export type: named (`export const foo`), default (`export default`), namespace (`export * as ns`), or all (`export *`)
`sourceLocation`	`{ line: number, column: number }`	Original location in source file

DynamicImport (represents `import()` expressions):

Field	Type	Description
`specifier`	`string | { type: 'TemplateLiteral' }`	The module specifier (string literal or template expression for dynamic paths)
`sourceLocation`	`{ line: number, column: number }`	Location in source file
`isStaticSpecifier`	`boolean`	Whether the specifier is a string literal (true) or expression (false)

The `ModuleNode` type (from our naming conventions) is populated with these extracted dependencies:

Field	Type	Description
`id`	`string`	Unique module identifier (initially the file path)
`filePath`	`string`	Absolute file system path
`dependencies`	`Map<string, string>`	Maps specifier → resolved module ID (filled during resolution)
`dependents`	`Set<string>`	Module IDs that import from this module (filled during graph building)
`exports`	`Set<string>`	Names of all exports this module provides
`imports`	`Map<string, string>`	Maps local import name → source specifier
`ast`	`object`	Full Abstract Syntax Tree
`source`	`string`	Original source code
`sideEffects`	`boolean`	Whether module has side effects (defaults to true)
`isEntry`	`boolean`	Whether this is an entry point
`usedExports`	`Set<string>`	Exports actually used by other modules (filled during tree shaking)

Parsing Algorithm

The parsing process follows a deterministic, step-by-step procedure that transforms source text into analyzable module data. This algorithm ensures consistent handling across different module formats and edge cases:

1. **Read Source Content**: Load the raw source code from the filesystem (or cache). Use the `FSCache.readFile(filePath)` method to benefit from caching and avoid redundant I/O operations. Preserve the exact source string for later code generation and source map construction.
2. **Detect Language and Syntax**: Examine the file extension (`.js` , `.jsx` , `.ts` , `.tsx`) and potentially the source content to determine the appropriate parsing mode. TypeScript files require TypeScript syntax support, while JSX files require JSX element parsing.
3. **Generate Abstract Syntax Tree**: Invoke the configured parser (`@babel/parser`) with appropriate plugins for the detected language features. Critical configuration includes:
    - `sourceType: 'module'` to parse ES module syntax
    - `plugins: ['jsx', 'typescript']` as needed for file type
    - `sourceFilename` set to the original file path for accurate source maps
    - `ranges: true` or `locations: true` to preserve position data for source maps and error reporting
4. **Traverse AST for Import Declarations**: Perform a depth-first traversal of the AST, specifically looking for nodes of type:
    - `ImportDeclaration` (ESM static imports: `import x from 'y'`)
    - `ExportNamedDeclaration` (named exports: `export { x }` or `export const x = 1`)
    - `ExportDefaultDeclaration` (default exports: `export default x`)
    - `ExportAllDeclaration` (namespace re-exports: `export * from 'y'`)
    - `CallExpression` with `callee.name === 'require'` (CommonJS: `require('x')`)
    - `ImportExpression` or `CallExpression` with `callee.type === 'Import'` (dynamic imports: `import('x')`)
5. **Extract and Normalize Specifiers**: For each dependency node, extract the module specifier string and normalize it:

    - Remove surrounding quotes (`'` , `"` , or backticks)
    - Preserve relative (`./`), absolute (`/`), or bare specifiers (`lodash`)
    - For template literals in dynamic imports (`import(`./${name}.js`)`), mark as non-static for resolution
6. **Build Import/Export Collections**: Create structured objects for each import/export:
    - For imports: Record the specifier, imported binding(s), and local alias(es)
    - For exports: Record exported name(s), local source (if re-export), and type
    - For dynamic imports: Record as potential code split points with location data
7. **Populate ModuleNode**: Create a new `ModuleNode` instance with:
    - `id` and `filePath` from the input
    - `ast` set to the generated AST
    - `source` set to the original source code
    - `imports` and `exports` populated from extracted data
    - `dependencies` initialized as an empty Map (to be filled during resolution)
    - `sideEffects` defaulted to `true` (can be updated later from package.json)
8. **Handle Parse Errors Gracefully**: If parsing fails (syntax errors, unsupported syntax), produce a descriptive error with file path, line number, and specific issue. Do not proceed with broken ASTs that would corrupt downstream analysis.

> **Key Insight**: The parser's primary responsibility is **extraction**, not interpretation. It identifies *what* dependencies exist and *where* they appear, but leaves *how* to resolve them (to file paths) and *whether* they're actually used (tree shaking) to later components. This separation of concerns enables caching (ASTs can be reused) and incremental processing.

ADR: Choosing an AST Parser

> **Decision: Use @babel/parser for JavaScript/TypeScript parsing**

Context: The bundler needs a reliable, accurate parser that can handle the full spectrum of JavaScript and TypeScript syntax, including modern ES2022+ features, JSX, Flow, and experimental proposals. The parser must produce a detailed AST with location information for source maps and must handle both ES modules and CommonJS syntax. Performance is important but secondary to correctness and maintainability in this educational context.

Options Considered:

1. **Acorn with plugins**: Lightweight, fast parser used by webpack and Rollup, requiring separate plugins for TypeScript, JSX, etc.
2. **@babel/parser (Babel Parser)**: Part of the Babel ecosystem, supports all modern JavaScript features and TypeScript via plugins, battle-tested.
3. **TypeScript Compiler API**: Native TypeScript parser that provides perfect TypeScript support and type information.

Decision: Use `@babel/parser` with appropriate plugins configured for the file type (.js, .jsx, .ts, .tsx).

Rationale:

- **Comprehensive syntax support**: `@babel/parser` handles the entire JavaScript syntax spectrum through versioned presets and supports TypeScript, JSX, Flow, and experimental proposals via plugins—all within a single, consistent API.
- **Ecosystem integration**: Babel's plugin ecosystem allows extensibility if we need to support additional syntax or custom transformations later.
- **Location data**: Provides excellent source location tracking (line/column) crucial for error reporting and source map generation.
- **Proven reliability**: Used by thousands of projects in production, including Babel itself, ensuring edge cases are handled.
- **TypeScript compromise**: While TypeScript's own parser offers better type analysis, `@babel/parser` provides sufficient syntax parsing for bundling purposes without requiring the full TypeScript compiler infrastructure.

Consequences:

- ✅ Single parser for both JavaScript and TypeScript with consistent output format
- ✅ Access to Babel's extensive plugin ecosystem for future extensions
- ✅ Good performance for our use case
- ⚠️ Additional dependency on Babel ecosystem

- ⚠️ No type information (only syntax), but sufficient for bundling
- ⚠️ Need to configure plugins appropriately for different file types

Comparison Table:

Option	Pros	Cons	Why Not Chosen
Acorn with plugins	Very fast, minimal footprint, used by production bundlers	Requires multiple plugins for full syntax support, less unified ecosystem	Good option but requires more integration work for TypeScript
@babel/parser	Complete syntax support, unified plugin system, excellent docs	Larger dependency, slightly slower than Acorn	**CHOSEN**: Best balance of completeness and maintainability
TypeScript Compiler API	Perfect TypeScript support, type information available	JavaScript-only parsing less robust, heavier dependency, complex API	Overkill for syntax parsing alone; better for type-aware transformations

Common Pitfalls

⚠️ Pitfall 1: Missing File Extension Assumptions

Description: Assuming import specifiers include file extensions (e.g., `import './utils.js'`) when most real-world code omits them (e.g., `import './utils'`).

Why it's wrong: Node.js and bundlers automatically try extensions (`.js`, `.ts`, `.jsx`) and `index.js` files. If the parser only records the literal specifier, the resolver will fail to find files without extensions.

Fix: The parser should extract the specifier exactly as written. The resolver component (not the parser) handles extension probing and directory/index file resolution. The parser's job is fidelity to source.

⚠️ Pitfall 2: Confusing Re-exports with Local Exports

Description: Treating `export { foo } from './bar'` (a re-export) the same as `export const foo = 1` (a local export) during export collection.

Why it's wrong: Re-exports don't create local bindings in the current module—they forward exports from another module. During tree shaking, re-exports require special handling: removing an unused re-export doesn't eliminate code, but removing a local export might.

Fix: Distinguish export types clearly in the `ExportSpecifier`:

- `type: 'named'` for local exports like `export const foo = 1`
- `type: 'namespace'` for `export * as ns from './module'`
- `type: 'all'` for `export * from './module'`
- Always include `source: string | null` field (null for local exports, specifier for re-exports)

⚠️ Pitfall 3: Ignoring Dynamic `import()` as Code-Split Points

Description: Only extracting static `import` statements and missing `import()` function calls, which represent potential code-split points.

Why it's wrong: Dynamic imports are the primary mechanism for code splitting in modern applications. Missing them means the bundler cannot create separate bundles for lazy-loaded modules.

Fix: Explicitly detect `ImportExpression` nodes (or `CallExpression` with `callee.type === 'Import'` in older AST formats). Record them separately from static imports with a flag indicating they're dynamic. Even if the specifier isn't a string literal (e.g., `import(`./${name}.js`)`), record it as a dynamic split point.

⚠️ Pitfall 4: Overlooking Side Effect Detection at Parse Time

Description: Focusing only on import/export statements and ignoring code that executes immediately (top-level function calls, assignments to globals) which may have side effects.

Why it's wrong: Tree shaking later needs to know if a module has side effects to determine if it can be removed when exports are unused. Pure parsing misses this.

Fix: While full side effect detection happens during tree shaking, the parser can note obvious side effects:

- Top-level function calls (`init()`)
- Assignments to global objects (`window.foo = bar`)

- This information supplements the `sideEffects` field from `package.json` but doesn't replace comprehensive analysis.

⚠️ Pitfall 5: Incorrect Handling of Import/Export Default Together

Description: Misinterpreting `import React, { Component } from 'react'` as two separate imports or mishandling `export { default as Foo } from './bar'`.

Why it's wrong: Default and named imports/exports have different runtime semantics. Incorrect representation breaks the runtime wrapper generation.

Fix: For imports, record `imported: 'default'` for default imports. For exports, distinguish `exported: 'default'` for default exports. Ensure the data structure supports mixed default and named imports/exports in a single statement.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
AST Parser	`@babel/parser` with basic plugins	`@babel/parser` with full plugin set + custom visitors
AST Traversal	Simple recursive visitor	`@babel/traverse` for efficiency and pattern matching
File Reading	Node.js `fs.readFile` with caching	`FSCache` class with TTL and memory limits
Module Node Creation	Factory function with explicit steps	Class with lazy parsing and caching

B. Recommended File/Module Structure

```
bundler/
├── src/
│   ├── index.ts                    # Main entry point
│   ├── parser/                     # This component
│   │   ├── index.ts                # Public API: parseModule, extractDependencies
│   │   ├── babel-config.ts         # Babel parser configuration
│   │   ├── visitors/               # AST visitor implementations
│   │   │   ├── import-visitor.ts   # Import statement extraction
│   │   │   ├── export-visitor.ts   # Export statement extraction
│   │   │   └── require-visitor.ts  # CommonJS require detection
│   │   ├── types.ts                # Type definitions (ImportSpecifier, etc.)
│   │   └── utils.ts                # Helper functions
│   ├── resolver/                   # Module resolution (next component)
│   ├── graph/                      # Module graph construction
│   ├── generator/                  # Code generation
│   └── utils/
│       ├── fs-cache.ts             # FSCache implementation
│       └── errors.ts               # Custom error types
└── package.json
```

C. Infrastructure Starter Code

Complete FSCache Implementation (prerequisite utility):

JAVASCRIPT

```javascript
// src/utils/fs-cache.ts

import fs from 'fs/promises';

import path from 'path';

import { fileURLToPath } from 'url';


/**

 * Cache entry for file system reads

 * @typedef {Object} CacheEntry

 * @property {string} content - File content

 * @property {number} timestamp - Last modified time

 * @property {string} filePath - Absolute file path

 */


/**

 * File System Cache with TTL and staleness checking

 */

export class FSCache {

  /**

   * @param {Object} options

   * @param {number} options.ttl - Time to live in milliseconds

   * @param {number} options.maxSize - Maximum cache entries

   */

  constructor(options = {}) {

    this.cache = new Map();

    this.ttl = options.ttl || 5000; // 5 seconds default

    this.maxSize = options.maxSize || 100;

  }


  /**

   * Read file with caching

   * @param {string} filePath - Absolute or relative file path

   * @returns {Promise<string>} File content

   */

  async readFile(filePath) {

    const absolutePath = path.resolve(filePath);



    // Check cache

    const cached = this.cache.get(absolutePath);
```

```
    if (cached && this._isValid(cached)) {
      return cached.content;
    }

    // Read fresh
    const content = await fs.readFile(absolutePath, 'utf-8');
    const stats = await fs.stat(absolutePath);

    // Create cache entry
    const entry = {
      content,
      timestamp: stats.mtimeMs,
      filePath: absolutePath
    };

    // Update cache (with LRU-like eviction if needed)
    this._setCache(absolutePath, entry);

    return content;
  }

  /**
   * Check if cache entry is still valid
   * @private
   * @param {CacheEntry} entry
   * @returns {boolean}
   */
  _isValid(entry) {
    const now = Date.now();
    return (now - entry.timestamp) < this.ttl;
  }

  /**
   * Set cache entry with size management
   * @private
   * @param {string} key
   * @param {CacheEntry} value
```

```
   */
  _setCache(key, value) {
    // Remove oldest entry if at max size
    if (this.cache.size >= this.maxSize) {
      const oldestKey = this.cache.keys().next().value;
      this.cache.delete(oldestKey);
    }

    // Remove existing entry to re-insert at end (LRU)
    this.cache.delete(key);
    this.cache.set(key, value);
  }

  /**
   * Clear entire cache
   */
  clear() {
    this.cache.clear();
  }

  /**
   * Invalidate specific file from cache
   * @param {string} filePath
   */
  invalidate(filePath) {
    const absolutePath = path.resolve(filePath);
    this.cache.delete(absolutePath);
  }
}
```

D. Core Logic Skeleton Code

Babel Parser Configuration:

JAVASCRIPT

```javascript
// src/parser/babel-config.ts

import * as parser from '@babel/parser';


/**

 * Get appropriate parser plugins based on file extension

 * @param {string} filePath

 * @returns {parser.ParserPlugin[]}

 */

export function getParserPlugins(filePath) {

  const plugins = [

    'jsx',

    'classProperties',

    'dynamicImport',

    'exportDefaultFrom',

    'exportNamespaceFrom',

    'decorators-legacy'

  ];



  if (filePath.endsWith('.ts') || filePath.endsWith('.tsx')) {

    plugins.push('typescript');

  }



  if (!filePath.endsWith('.tsx') && !filePath.endsWith('.jsx')) {

    // Remove jsx plugin for non-JSX files if desired

    const jsxIndex = plugins.indexOf('jsx');

    if (jsxIndex > -1) plugins.splice(jsxIndex, 1);

  }



  return plugins;

}


/**

 * Parse source code with appropriate configuration

 * @param {string} sourceCode

 * @param {string} filePath

 * @returns {parser.ParseResult<import('@babel/types').File>}

 */
```

```
export function parseSource(sourceCode, filePath) {
  const plugins = getParserPlugins(filePath);


  return parser.parse(sourceCode, {
    sourceType: 'module',
    sourceFilename: filePath,
    plugins,
    ranges: true,  // For source map generation
    allowImportExportEverywhere: false,
    allowReturnOutsideFunction: false,
    allowSuperOutsideMethod: false,
    allowUndeclaredExports: false,
    errorRecovery: false, // Fail fast on syntax errors
  });
}
```

Main Parser Function with TODOs:

JAVASCRIPT

```javascript
// src/parser/index.ts

import { parseSource } from './babel-config.js';

import { ImportVisitor } from './visitors/import-visitor.js';

import { ExportVisitor } from './visitors/export-visitor.js';


/**

 * Parse module and extract dependencies

 * @param {string} sourceCode - Raw source content

 * @param {string} filePath - Absolute path to source file

 * @returns {Promise<{

 *   ast: import('@babel/types').File,

 *   imports: Array<ImportSpecifier>,

 *   exports: Array<ExportSpecifier>,

 *   dynamicImports: Array<DynamicImport>

 * }>}

 */

export async function parseModule(sourceCode, filePath) {

  // TODO 1: Parse source code using parseSource() function

  // This will throw on syntax errors - let it propagate

  const ast = parseSource(sourceCode, filePath);



  // TODO 2: Initialize empty collections for imports, exports, dynamic imports

  const imports = [];

  const exports = [];

  const dynamicImports = [];



  // TODO 3: Traverse AST for ImportDeclaration nodes

  // For each import node:

  //   - Extract specifier (remove quotes)

  //   - Determine if it's default, namespace, or named imports

  //   - For each imported binding, create an ImportSpecifier

  //   - Add to imports array

  ast.program.body.forEach(node => {

    if (node.type === 'ImportDeclaration') {

      // Process import statement

    }

  });
```

```
  // TODO 4: Traverse AST for ExportDeclaration nodes
  // Handle: ExportNamedDeclaration, ExportDefaultDeclaration, ExportAllDeclaration
  // For each export type:
  //   - Determine export type (named, default, namespace, all)
  //   - Extract local name (if any) and exported name
  //   - For re-exports, extract source module specifier
  //   - Create ExportSpecifier and add to exports array

  // TODO 5: Traverse entire AST for CallExpression nodes
  // Look for: require('specifier') calls (CommonJS)
  // Look for: import('specifier') calls (dynamic imports)
  // For require(): treat as static dependency but with CommonJS semantics
  // For import(): create DynamicImport record with location info

  // TODO 6: Return structured result
  return {
    ast,
    imports,
    exports,
    dynamicImports
  };
}

/**
 * Extract dependencies from an existing AST (cached version)
 * @param {import('@babel/types').File} ast
 * @returns {{
 *   imports: Array<ImportSpecifier>,
 *   exports: Array<ExportSpecifier>,
 *   dynamicImports: Array<DynamicImport>
 * }}
 */
export function extractDependencies(ast) {
  // TODO 1: Implement logic similar to parseModule but using existing AST
  // TODO 2: Reuse the same visitor/traversal logic
  // TODO 3: Return only the dependency collections (not the AST)
```

```
  return {
    imports: [],
    exports: [],
    dynamicImports: []
  };
}

/**
 * Factory function to create a ModuleNode from source
 * @param {string} filePath
 * @param {string} sourceCode
 * @param {boolean} isEntry
 * @returns {Promise<ModuleNode>}
 */
export async function createModuleNode(filePath, sourceCode, isEntry = false) {
  // TODO 1: Parse module and extract dependencies using parseModule()
  const { ast, imports, exports, dynamicImports } = await parseModule(sourceCode, filePath);


  // TODO 2: Initialize a new ModuleNode object
  const moduleNode = {
    id: filePath,  // Will be normalized later
    filePath,
    dependencies: new Map(),  // Will be filled during resolution
    dependents: new Set(),    // Will be filled during graph building
    exports: new Set(),
    imports: new Map(),
    ast,
    source: sourceCode,
    sideEffects: true,  // Default assumption
    isEntry,
    usedExports: new Set()
  };


  // TODO 3: Populate imports Map: localName -> specifier
  imports.forEach(imp => {
    moduleNode.imports.set(imp.local, imp.specifier);
  });
```

```
  // TODO 4: Populate exports Set: all exported names
  exports.forEach(exp => {
    if (exp.exported) {
      moduleNode.exports.add(exp.exported);
    }
  });

  // TODO 5: Note dynamic imports as special dependencies
  // These will be processed during code splitting phase

  return moduleNode;
}
```

AST Visitor Implementation Skeleton:

```javascript
// src/parser/visitors/import-visitor.js

/**
 * Extract import statements from AST nodes
 * @param {import('@babel/types').ImportDeclaration} node
 * @returns {Array<ImportSpecifier>}
 */
export function processImportDeclaration(node) {
  const specifier = node.source.value;
  const imports = [];

  // TODO 1: Handle default import: import React from 'react'
  // Creates: { specifier, type: 'static', imported: 'default', local: 'React' }

  // TODO 2: Handle namespace import: import * as React from 'react'
  // Creates: { specifier, type: 'static', imported: '*', local: 'React' }

  // TODO 3: Handle named imports: import { Component } from 'react'
  // Creates: { specifier, type: 'static', imported: 'Component', local: 'Component' }

  // TODO 4: Handle named imports with alias: import { Component as Comp } from 'react'
  // Creates: { specifier, type: 'static', imported: 'Component', local: 'Comp' }

  // TODO 5: Handle mixed imports: import React, { Component } from 'react'
  // Creates two separate ImportSpecifier records

  return imports;
}
```

E. Language-Specific Hints

- **Use async/await for file operations**: Node.js `fs/promises` provides promise-based file operations that work well with async/await syntax.
- **Leverage ES6 Maps and Sets**: Use `Map` for dependencies (specifier → moduleId) and `Set` for exports/usedExports for O(1) lookups and automatic deduplication.
- **Preserve source locations**: Babel parser's `ranges: true` option gives you `[start, end]` character ranges for each node, crucial for source maps.
- **Handle TypeScript syntax carefully**: Remember that TypeScript's `import type` and `export type` statements should be ignored for bundling purposes—they're erased at runtime.
- **Consider using `@babel/traverse`**: For complex AST traversal, the `@babel/traverse` package provides efficient visitor pattern implementation, but start simple with manual traversal for clarity.

F. Milestone Checkpoint

After implementing the Parser & Dependency Extractor, verify correctness with this test:

```bash
# Create a test file
cat > test-module.js << 'EOF'
import { foo } from './utils';
import React from 'react';
export const bar = 'baz';
export default function main() {};
export { default as reexport } from './other';
EOF

# Run your parser (adjust path based on your structure)
node -e "
const { createModuleNode } = require('./src/parser/index.js');
const fs = require('fs');
const source = fs.readFileSync('./test-module.js', 'utf-8');
createModuleNode('./test-module.js', source, true).then(module => {
  console.log('Imports:', Array.from(module.imports.entries()));
  console.log('Exports:', Array.from(module.exports));
  console.log('Has AST?', !!module.ast);
});
"
```

Expected Output:

- `Imports` should show: `[['foo', './utils'], ['React', 'react']]`
- `Exports` should include: `['bar', 'default', 'reexport']`
- `module.ast` should be a valid AST object
- No syntax errors should be thrown

Signs of Problems:

- Missing imports or exports → traversal logic incomplete
- Syntax errors on valid ES2020+ code → missing parser plugins
- Incorrect paths in output → not stripping quotes from specifiers
- Crash on TypeScript files → missing TypeScript plugin configuration

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Unexpected token" errors	Missing parser plugin for syntax (JSX, TypeScript, decorators)	Check file extension and compare with `getParserPlugins()` output	Add appropriate plugin to configuration
Import statements not detected	Traversal missing node types or visiting wrong AST level	Log all node types during traversal, verify you're checking `program.body`	Add handling for `ImportDeclaration`, `ExportNamedDeclaration`, etc.
Specifiers include quotes	Not stripping quotes from string literals	Check raw node: `node.source.value` vs `node.source.raw`	Use `node.source.value` which is already unquoted
Default vs named import confusion	Misinterpreting import specifier structure	Log full import node structure: `node.specifiers` array	Examine `specifier.type`: `ImportDefaultSpecifier`, `ImportNamespaceSpecifier`, `ImportSpecifier`
Dynamic imports not found	Looking for wrong node type	Log all `CallExpression` nodes, check `callee.type`	Look for `ImportExpression` (new) or `CallExpression` with `callee.type === 'Import'`
Re-exports creating local exports	Not checking `node.source` field on exports	Check if `ExportNamedDeclaration` has `node.source`	If `node.source` exists, it's a re-export, not local

Component 2: Module Resolver

> **Milestone(s):** Milestone 2

The **Module Resolver** is the navigational heart of our bundler, responsible for the critical task of translating abstract module references into concrete file system paths. While the parser identifies *what* modules a file needs, the resolver determines *where* those modules physically exist on disk—a non-trivial problem given JavaScript's multiple module systems and Node.js's complex resolution rules.

Mental Model: The Address Translator

Imagine you're a postal service handling letters with various addressing formats. Some letters have exact street addresses (`./components/Button.js`), others have only building names (`react`), and some reference buildings by department (`lodash/map`). Your job is to transform every address into a precise, deliverable location—finding exactly which building, which floor, and which room to deliver to.

The Module Resolver operates similarly:

- **Relative/absolute paths** are like street addresses—you follow the path from the current location
- **Bare specifiers** (like `react`) are like building names—you search through known directories (`node_modules`)
- **Package subpaths** (like `lodash/map`) require checking the building's directory map (`package.json` exports field) to find the right room

This translation from abstract to concrete is essential because while developers think in logical imports, the bundler must work with physical files.

Interface

The resolver exposes a single primary function with a clear contract:

Method	Parameters	Returns	Description
`resolve(specifier, fromDir)`	`specifier`: string `fromDir`: string (absolute path)	`Promise<string>` (absolute path)	Resolves a module specifier relative to a directory. Throws if module cannot be found.
`resolveSync(specifier, fromDir)`	Same as above	`string` (absolute path)	Synchronous version for performance-critical paths

The resolver also maintains internal state for caching and configuration:

Internal State	Type	Purpose
`fsCache`	`FSCache` instance	Caches file reads (especially `package.json`) to avoid repeated I/O
`resolutionCache`	`Map<string, string>`	Maps `specifier+fromDir` → resolved path to avoid re-computation
`extensions`	`string[]`	Default: `['.js', '.ts', '.jsx', '.tsx', '.json']`
`mainFields`	`string[]`	Default: `['module', 'main']` (package.json fields to check)

Resolution Algorithm

The resolver implements a multi-stage algorithm that closely follows Node.js's CommonJS resolution algorithm, with extensions for ESM. The process is deterministic and follows this priority order:

1. Path Classification

The resolver first categorizes the specifier type:

Specifier Pattern	Type	Resolution Strategy
`./foo` or `../bar`	Relative	Resolve relative to `fromDir`
`/absolute/path`	Absolute	Use as-is (with extension resolution)
`#internal/package`	Package internal	Handle via `package.json` imports field
Starts with letter	Bare/bare specifier	Node modules resolution

2. Core Resolution Steps

For each specifier, the resolver follows this numbered procedure:

1. **Normalize specifier**: Remove any leading `./` or `../` traversal, convert backslashes to forward slashes
2. **Check cache**: If `specifier + fromDir` exists in `resolutionCache`, return cached result
3. **Determine resolution type**:
   - If specifier is absolute or relative → **Path resolution** (step 4)
   - Otherwise → **Bare specifier resolution** (step 5)
4. **Path resolution (relative/absolute)**:
   1. Join `fromDir` with specifier (if relative) to get candidate path
   2. If candidate is a file → return it
   3. If candidate is a directory → look for `package.json` (step 6) or `index` file
   4. Try adding extensions from `extensions` list
   5. If all fail, throw "Module not found"
5. **Bare specifier resolution**:
   1. Starting at `fromDir`, traverse up parent directories looking for `node_modules`
   2. At each `node_modules` directory, check for `specifier` as subdirectory
   3. If found → examine its `package.json` (step 6)
   4. If not found, move to parent directory and repeat until filesystem root

5. If root reached without finding → throw "Module not found"

6. **Package entry point resolution** (when specifier points to a package directory):
    1. Read `package.json` from package directory (cached via `fsCache`)
    2. Check fields in `mainFields` order (e.g., `module` then `main`)
    3. If `exports` field exists → handle conditional exports (step 7)
    4. If no field matches, default to `index.js` in package root
    5. Resolve the found entry relative to package directory

7. **Conditional exports handling** (simplified version):
    1. Parse `exports` field (can be string, object, or array)
    2. Check for `.` (main entry) or subpath keys matching specifier
    3. Evaluate conditions (priority: `import` → `require` → `default`)
    4. Return first matching export target

3. Extension Resolution Logic

When a path doesn't exist as-is, the resolver tries these variations:

```javascript
// For specifier "./utils":

// 1. Check "./utils" (as directory with package.json)

// 2. Check "./utils.js"

// 3. Check "./utils.ts"

// 4. Check "./utils/index.js"

// 5. Check "./utils/index.ts"
```

This order matches Node.js behavior and ensures compatibility with both explicit extensions and extensionless imports.

ADR: Implementing Node Resolution

> **Decision: Hybrid Resolution Implementation**
>
> - **Context**: Our educational bundler must accurately resolve modules according to Node.js rules, which are notoriously complex (40+ edge cases in the official spec). We need balance between educational value and implementation completeness.
> - **Options Considered**:
>   1. **Implement full algorithm from Node.js spec** (~1000+ lines with all edge cases)
>   2. **Use the mature `resolve` npm package** (battle-tested, handles all edge cases)
>   3. **Implement core algorithm + fallback to `resolve`** (educational for common cases, robust for edge cases)
> - **Decision**: Option 3 – Implement the core resolution algorithm ourselves for learning, but use the `resolve` package as a fallback for complex cases (conditional exports, symlinks).
> - **Rationale**:
>   - Implementing the full algorithm is educational overkill—the value is in understanding the *concepts*, not every edge case
>   - The `resolve` package is 7+ years mature, handles symlinks, `exports` field, and Windows paths correctly
>   - By implementing the core (80% of cases), learners understand the algorithm; by delegating edge cases, they get a working system
>   - This follows the "learning scaffolding" principle: build understanding gradually
> - **Consequences**:
>   - Positive: Faster implementation, fewer bugs in edge cases, focus on educational value
>   - Negative: Adds a dependency, slight abstraction leak when `resolve` is called
>   - Mitigation: Document when we use `resolve` and why, maintain clear boundaries

Option	Pros	Cons	Chosen?
Full custom implementation	Complete control, maximal learning	High complexity, many bugs, time-consuming	No
Use `resolve` package only	Robust, production-ready, minimal code	Minimal learning value, black box	No
Hybrid approach	**Best of both: learn core + robustness**	**Slight complexity in fallback logic**	**Yes**

Common Pitfalls

⚠️ Pitfall: Infinite loops with symlinks

- **Description**: When `node_modules` contains symlinks to parent directories, naive traversal can loop forever
- **Why it's wrong**: The resolver gets stuck checking the same directories repeatedly, causing infinite recursion or stack overflow
- **Fix**: Maintain a `visited` set of realpath-normalized directories, skip already-visited paths

⚠️ Pitfall: Misordering `package.json` field priority

- **Description**: Checking `main` before `module`, or not respecting the `exports` field priority
- **Why it's wrong**: Breaks packages optimized for bundlers (which use `module` for ESM), or breaks new packages using `exports`
- **Fix**: Use field order `['module', 'main']` for bundlers (ESM-first), respect Node's `exports` precedence rules

⚠️ Pitfall: Ignoring `sideEffects: false` during resolution

- **Description**: Not reading the `sideEffects` field during package resolution, only during tree shaking
- **Why it's wrong**: Missing early optimization opportunity—can't skip entire packages during graph building
- **Fix**: Parse `sideEffects` when reading `package.json`, attach to `ModuleNode` for later use

⚠️ Pitfall: Case-insensitive file systems on macOS/Windows

- **Description**: `require('React')` works on macOS even though package is `react`, causing cross-platform bugs
- **Why it's wrong**: Bundle works on developer machine but fails on Linux CI servers
- **Fix**: Use case-sensitive comparison, or normalize to lowercase and check both cases

⚠️ Pitfall: Circular symlinks in `node_modules`

- **Description**: Packages that symlink to each other (`A → B → A`) causing infinite resolution
- **Why it's wrong**: Similar to directory symlinks but harder to detect
- **Fix**: Track package IDs (name+version) in addition to paths, detect cycles

Resolution Example Walkthrough

Let's trace a concrete example to see the algorithm in action:

Scenario: File `/project/src/app.js` imports `import { Button } from './components/Button'`

1. **Classification**: `./components/Button` is a relative specifier
2. **Path joining**: Join `/project/src` + `./components/Button` = `/project/src/components/Button`
3. **File check**: `/project/src/components/Button` doesn't exist as file
4. **Extension trial**: Try `/project/src/components/Button.js` → exists! Return this path

Second scenario: Same file imports `import React from 'react'`

1. **Classification**: `react` is a bare specifier
2. **Start at `/project/src`**: Check `/project/src/node_modules/react` → not found
3. **Move to parent**: Check `/project/node_modules/react` → found!
4. **Read package.json**: `/project/node_modules/react/package.json`
5. **Check fields**: `module` field = `index.js`, `main` field = `index.js`
6. **Resolve**: `/project/node_modules/react/index.js` → exists, return this

Third scenario (complex): Import `import map from 'lodash/map'`

1. **Bare with subpath**: `lodash/map` means package `lodash`, subpath `map`

2. **Find package**: Locate `/project/node_modules/lodash`
3. **Check exports**: `package.json` has `exports` field mapping `./map` → `./map.js`
4. **Resolve**: `/project/node_modules/lodash/map.js` → exists, return

This walkthrough shows how different specifier types trigger different resolution paths, all converging to absolute file paths.

Data Flow with Other Components

The resolver doesn't operate in isolation—it's part of a larger data flow:

```
Parser (extracts specifiers)
    ↓
Resolver (translates specifier → filePath)
    ↓
GraphBuilder (creates ModuleNode with filePath)
    ↓
Parser (reads file at filePath, extracts more specifiers)
```

This creates a recursive discovery process: parse → resolve → parse more → resolve more, until the entire dependency graph is built.

The resolver also interacts with the `FSCache` to avoid redundant filesystem operations:

```javascript
// Pseudocode for the readPackageJson helper

async function readPackageJson(dir) {

  const pkgPath = path.join(dir, 'package.json');

  return await fsCache.readFile(pkgPath); // Cached read

}
```

Error Handling Strategy

The resolver must provide clear, actionable error messages:

Error Condition	Error Message Format	Recovery Suggestion
Module not found	`Cannot find module '${specifier}' from '${fromDir}'`	Check if package is installed, typo in specifier
Missing `package.json`	`No package.json found in ${pkgDir}`	Corrupted package installation
Invalid `exports` field	`Invalid exports field in ${pkgPath}: ${error}`	Check package.json syntax
Circular symlink	`Circular symlink detected: ${realPath}`	Clean node_modules and reinstall

The resolver follows a "fail-fast" strategy: any unresolved module stops the bundling process immediately, as a missing dependency makes the bundle unusable.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
File system access	Node.js `fs/promises`	Custom `FSCache` with TTL
Path manipulation	Node.js `path` module	Custom normalized path utils
Package.json parsing	`JSON.parse` with error handling	Schema validation with Zod/Joi
Fallback resolver	`require.resolve` (limited)	`resolve` npm package

Recommended File Structure

```
bundler/
├── src/
│   ├── resolver/
│   │   ├── index.js              # Main resolver interface
│   │   ├── ResolutionContext.js  # Resolution state and cache
│   │   ├── pathResolution.js     # Relative/absolute resolution
│   │   ├── nodeModules.js        # Bare specifier resolution
│   │   ├── packageJson.js        # package.json reading and parsing
│   │   └── errors.js             # Resolution-specific errors
│   ├── fs/
│   │   └── FSCache.js            # Shared file system cache
│   └── utils/
│       └── pathUtils.js          # Path normalization helpers
```

Infrastructure Starter Code

Here's a complete, ready-to-use `FSCache` implementation that the resolver can share with other components:

JAVASCRIPT

```javascript
// src/fs/FSCache.js

import fs from 'fs/promises';

import path from 'path';

import { fileURLToPath } from 'url';


/**

 * Cache entry for file system reads

 * @typedef {Object} CacheEntry

 * @property {string} content - File content

 * @property {number} timestamp - Last modified time

 * @property {string} filePath - Original file path

 */


/**

 * Simple TTL-based file system cache

 */

export class FSCache {

  /**

   * @param {number} ttlMs - Time to live in milliseconds

   */

  constructor(ttlMs = 5000) {

    /** @type {Map<string, CacheEntry>} */

    this.cache = new Map();

    this.ttl = ttlMs;

  }


  /**

   * Read file with caching

   * @param {string} filePath - Absolute path to file

   * @returns {Promise<string>} File content

   */

  async readFile(filePath) {

    const normalizedPath = path.normalize(filePath);

    const now = Date.now();



    // Check cache

    const cached = this.cache.get(normalizedPath);

    if (cached && (now - cached.timestamp) < this.ttl) {
```

```
      return cached.content;
    }

    // Read fresh
    try {
      const content = await fs.readFile(normalizedPath, 'utf-8');
      const stats = await fs.stat(normalizedPath);

      const entry = {
        content,
        timestamp: stats.mtimeMs,
        filePath: normalizedPath
      };

      this.cache.set(normalizedPath, entry);
      return content;
    } catch (error) {
      throw new Error(`Failed to read ${normalizedPath}: ${error.message}`);
    }
  }

  /**
   * Read and parse JSON file with caching
   * @param {string} filePath - Absolute path to JSON file
   * @returns {Promise<any>} Parsed JSON content
   */
  async readJson(filePath) {
    const content = await this.readFile(filePath);
    try {
      return JSON.parse(content);
    } catch (error) {
      throw new Error(`Invalid JSON in ${filePath}: ${error.message}`);
    }
  }

  /**
   * Clear cache for specific file or all files
```

```
   * @param {string} [filePath] - Optional specific file to clear
   */
  clear(filePath) {
    if (filePath) {
      this.cache.delete(path.normalize(filePath));
    } else {
      this.cache.clear();
    }
  }
}
```

Core Logic Skeleton Code

Here's the main resolver implementation with detailed TODOs following the algorithm steps:

JAVASCRIPT

```javascript
// src/resolver/ResolutionContext.js

import path from 'path';

import { FSCache } from '../fs/FSCache.js';


export class ResolutionContext {

  /**

   * @param {Object} options

   * @param {string[]} options.extensions - File extensions to try

   * @param {string[]} options.mainFields - package.json fields to check

   * @param {boolean} options.preserveSymlinks - Whether to resolve symlinks

   */

  constructor(options = {}) {

    this.extensions = options.extensions || ['.js', '.ts', '.jsx', '.tsx', '.json'];

    this.mainFields = options.mainFields || ['module', 'main'];

    this.preserveSymlinks = options.preserveSymlinks || false;



    /** @type {FSCache} */

    this.fsCache = new FSCache();



    /** @type {Map<string, string>} */

    this.resolutionCache = new Map();



    /** @type {Set<string>} */

    this.visitedSymlinks = new Set();

  }


  /**

   * Generate cache key for specifier + fromDir

   */

  _cacheKey(specifier, fromDir) {

    return `${specifier}::${fromDir}`;

  }


  /**

   * Main resolution method

   * @param {string} specifier - Module specifier to resolve

   * @param {string} fromDir - Absolute directory to resolve from

   * @returns {Promise<string>} Absolute path to resolved module
```

```
   */
  async resolve(specifier, fromDir) {
    // TODO 1: Normalize inputs
    //   - Ensure fromDir is absolute (path.isAbsolute)
    //   - Normalize specifier (remove leading ./ if present)

    // TODO 2: Check cache
    //   - Generate cache key with _cacheKey()
    //   - If found in resolutionCache, return cached value

    let resolvedPath;

    // TODO 3: Classify specifier type
    //   - If specifier starts with '.' or '/' or contains ':' → path resolution
    //   - Otherwise → bare specifier resolution

    if (/* is path specifier */) {
      // TODO 4: Handle path resolution
      //   - Call _resolvePath(specifier, fromDir)
    } else {
      // TODO 5: Handle bare specifier resolution
      //   - Call _resolveBareSpecifier(specifier, fromDir)
    }

    // TODO 6: Cache result before returning
    //   - Store in resolutionCache
    //   - Return resolvedPath
  }

  /**
   * Resolve relative or absolute path specifier
   */
  async _resolvePath(specifier, fromDir) {
    // TODO 1: Join paths
    //   - If specifier is absolute, use as-is
    //   - If relative, join with fromDir
```

```
    // TODO 2: Try as file
    //   - Check if candidate exists as file (with fs.stat)
    //   - If yes, return candidate

    // TODO 3: Try with extensions
    //   - For each extension in this.extensions:
    //     - candidate = base + extension
    //     - Check if exists as file
    //     - If yes, return candidate

    // TODO 4: Try as directory
    //   - Check if candidate is directory
    //   - If yes, read package.json (if exists) for "main" field
    //   - Otherwise try candidate/index.js, candidate/index.ts, etc.

    // TODO 5: If all fail, throw "Module not found" error
  }

  /**
   * Resolve bare specifier (like 'react' or 'lodash/map')
   */
  async _resolveBareSpecifier(specifier, fromDir) {
    // TODO 1: Parse package name and subpath
    //   - If specifier contains '/', split into [packageName, ...subpathParts]
    //   - Handle @scoped/package names

    // TODO 2: Find package directory
    //   - Call _findPackageDir(packageName, fromDir)
    //   - This walks up parent directories looking for node_modules/packageName

    // TODO 3: Read package.json
    //   - Use this.fsCache.readJson() to read package.json from package dir

    // TODO 4: Resolve package entry
    //   - If subpath exists, handle via exports field or direct file lookup
    //   - Otherwise, use main fields (this.mainFields) to find entry
    //   - Fallback to index.js
```

```
  // TODO 5: Join and return final path
  //   - Join package dir with resolved entry path
  //   - Call _resolvePath() to handle final resolution (extensions, etc.)
}

/**
 * Find package directory by walking up node_modules
 */
async _findPackageDir(packageName, startDir) {
  // TODO 1: Initialize currentDir = startDir

  // TODO 2: While currentDir not at filesystem root:
  //   - Construct nodeModulesPath = path.join(currentDir, 'node_modules', packageName)
  //   - Check if directory exists
  //   - If yes, return nodeModulesPath
  //   - Otherwise, move to parent directory: currentDir = path.dirname(currentDir)

  // TODO 3: If root reached without finding, throw "Package not found"

  // BONUS TODO: Handle symlinks
  //   - Use fs.realpath() if this.preserveSymlinks is false
  //   - Track visited realpaths in this.visitedSymlinks to avoid cycles
}

/**
 * Parse package.json exports field (simplified)
 */
async _resolveExports(pkg, subpath, packageDir) {
  // TODO 1: If no exports field, return null

  // TODO 2: If exports is string, use it for '.' (main entry) only

  // TODO 3: If exports is object:
  //   - Build full subpath (e.g., './map' for lodash/map)
  //   - Look for exact match in exports keys
  //   - Handle conditional exports (import/require/default)
```

```
    //   - Return matched value


    // TODO 4: Normalize export target (remove leading './')
    //   - Join with packageDir and return


    // Note: Full exports handling is complex; for learning, implement basic cases
    // and use the 'resolve' package fallback for advanced cases
  }
}
```

Language-Specific Hints

1. **Use `path.isAbsolute()`** not string startsWith `/` for cross-platform compatibility
2. **Normalize paths with `path.normalize()`** before caching or comparison
3. **Use `fs.realpath()`** to resolve symlinks, but cache results to avoid performance hit
4. **Handle JSON parse errors** gracefully—malformed `package.json` should throw clear error
5. **Consider using `import.meta.resolve()`** (Node 20+) as reference implementation check

Milestone Checkpoint

After implementing the resolver, verify it works:

```bash
# Test with a simple project
mkdir test-resolver
cd test-resolver
npm init -y
npm install react

# Create test file
cat > test.js << 'EOF'
const { resolve } = require('../src/resolver');

async function test() {
  // Test relative resolution
  console.log(await resolve('./test.js', process.cwd()));

  // Test node_modules resolution
  console.log(await resolve('react', process.cwd()));

  // Test subpath
  console.log(await resolve('react/jsx-runtime', process.cwd()));
}

test().catch(console.error);
EOF

node test.js
```

Expected Output:

- First line: Absolute path to `test.js` in current directory
- Second line: Path to `react`'s main entry (e.g., `.../node_modules/react/index.js`)
- Third line: Path to `react/jsx-runtime` module

Signs of Problems:

- `Module not found` errors: Check `node_modules` existence, typo in specifier
- Wrong file resolved: Check extension order, package.json field priority
- Infinite loop: Likely symlink issue—add visited set tracking

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Module not found" for installed package	Wrong `node_modules` traversal	Log each directory checked in `_findPackageDir`	Ensure traversal goes up to correct parent
Wrong file extension resolved	Incorrect extension order	Check `this.extensions` order and trial logic	Match Node.js order: `.js` before `.ts` before `.json`
Package subpath resolves incorrectly	Missing `exports` field handling	Log package.json content and exports parsing	Implement basic exports field support
Resolution extremely slow	No caching, repeated I/O	Add cache hit/miss logging	Implement `FSCache` and `resolutionCache`
Works on Mac, fails on Linux	Case sensitivity issue	Log actual vs expected path casing	Use case-sensitive comparison or normalize

Component 3: Graph Builder & Transformer

> **Milestone(s):** Milestone 1, Milestone 3, Milestone 4

The **Graph Builder & Transformer** is the analytical core of our bundler, responsible for constructing the complete dependency graph, performing static analysis for tree shaking, and preparing modules for final code generation. This component takes the raw materials discovered by the parser and resolver—individual modules with their dependencies—and weaves them into a coherent, optimized structure ready for bundling.

Mental Model: The Dependency Cartographer

Imagine you're tasked with mapping an archipelago of islands. Each island (module) has bridges (imports) connecting to other islands. Some islands import supplies from others, while some export goods (exports) for others to use. Your job as a cartographer has three phases:

1. **Surveying**: Starting from your home island (entry point), you systematically explore every connected island via bridges, recording each island's layout, what it exports, and which other islands it connects to. You continue until you've mapped every reachable island in the archipelago.
2. **Resource Assessment**: You now need to decide which islands are essential for your colony's survival. You trace supply chains starting from your home island—only the goods (exports) that actually reach your home island through the import network are marked as "essential." Islands that export nothing essential, and aren't involved in critical infrastructure (side effects), can be left off the final map.
3. **Route Planning**: Finally, you determine the optimal order to visit islands when delivering supplies. Dependencies must be visited before dependents—you can't deliver goods from an island until that island has received its own imported supplies first. This creates a logistical sequence for the entire supply chain.

This mental model captures the three core responsibilities of the Graph Builder & Transformer: building the complete module graph, marking used exports for tree shaking, and topologically sorting modules for correct execution order.

Interface

The component centers around the `ModuleGraph` class, which manages the entire dependency graph and provides methods for construction, analysis, and transformation. The following table details its complete interface:

ModuleGraph Class Interface

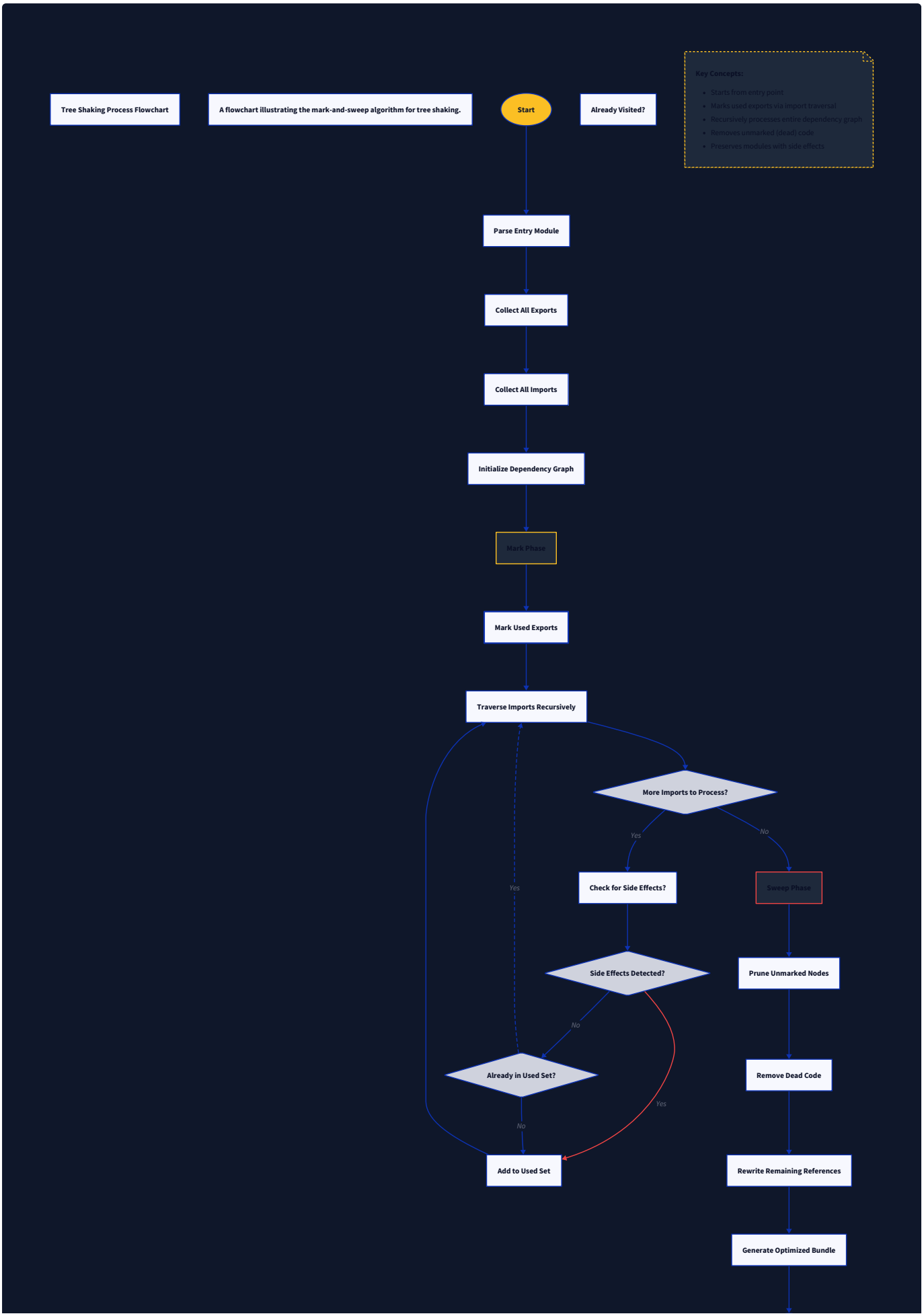
Method	Parameters	Returns	Description
`addModule(module)`	`module: ModuleNode`	`void`	Adds a module to the graph, ensuring no duplicates by `id`. Throws error if module with same ID already exists.
`getModule(id)`	`id: string`	`ModuleNode` or `undefined`	Retrieves a module by its unique identifier (typically the resolved absolute path).
`linkDependency(importerId, specifier, dependencyId)`	`importerId: string`, `specifier: string`, `dependencyId: string`	`void`	Creates bidirectional links between two modules. Updates the importer's `dependencies` map and the dependency's `dependents` set.
`build(entryPaths)`	`entryPaths: string[]`	`Promise<ModuleGraph>`	High-level orchestration method that constructs the complete graph starting from entry modules. Returns the graph itself for method chaining.
`markUsedExports(entryModuleId)`	`entryModuleId: string`	`void`	Performs static analysis to determine which exports are actually used, starting from the given entry module. Marks used exports in each `ModuleNode.usedExports`.
`shake()`	None	`ModuleGraph`	Removes unused modules and unused exports from the graph based on the markings from `markUsedExports`. Returns the graph for chaining.
`topologicalSort()`	None	`ModuleNode[]`	Returns all modules in the graph sorted in **post-order dependency order** (dependencies before dependents). Essential for correct bundle generation.

ModuleNode Methods Used in Graph Building

Method	Parameters	Returns	Description
`addDependency(specifier, moduleId)`	`specifier: string`, `moduleId: string`	`void`	Records a dependency from this module to another. Stores the mapping from import specifier to resolved module ID in the `dependencies` map.
`addExport(exportName)`	`exportName: string`	`void`	Adds an export name to this module's public interface. For default exports, uses the special name `"default"`.
`markExportUsed(exportName)`	`exportName: string`	`void`	Marks a specific export as "used" during the tree shaking analysis. Updates the `usedExports` set.
`shouldIncludeInBundle()`	None	`boolean`	Determines if this module should be included in the final bundle. Returns `true` if: 1) Module is an entry point, OR 2) Any of its exports are used, OR 3) Module has side effects (`sideEffects: true` or no `sideEffects` field).

Graph Construction & Shaking Algorithm

The graph building and transformation process follows a rigorous, multi-stage algorithm that ensures correctness while enabling aggressive dead code elimination. The complete process is visualized in the tree shaking flowchart:


Tree Shaking Process Flowchart
A flowchart illustrating the mark-and-sweep algorithm for tree shaking.
Start
Already Visited?
Key Concepts:
Starts from entry point
Marks used exports via import traversal
Recursively processes entire dependency graph
Removes unmarked (dead) code
Preserves modules with side effects
Parse Entry Module
Collect All Exports
Collect All Imports
Initialize Dependency Graph
Mark Phase
Mark Used Exports
Traverse Imports Recursively
More Imports to Process?
Yes
No
Check for Side Effects?
Sweep Phase
Yes
Side Effects Detected?
Prune Unmarked Nodes
No
Already in Used Set?
Remove Dead Code
Yes
No
Add to Used Set
Rewrite Remaining References
Generate Optimized Bundle

Stage 1: Graph Construction via BFS from Entry Points

The first stage builds the complete **transitive closure** of all modules reachable from the entry points.

1. **Initialize**: Create an empty `ModuleGraph` instance and a queue for breadth-first traversal.
2. **Process Entry Points**: For each entry path in `entryPaths`:
   - Resolve the path to an absolute file path using the Module Resolver (Component 2).
   - Read the file content using `FSCache.readFile()`.
   - Parse the module using `parseModule()` (from Component 1) to create a `ModuleNode` with `isEntry: true`.
   - Add the module to the graph via `ModuleGraph.addModule()`.
   - Enqueue the module for dependency exploration.
3. **BFS Traversal**: While the queue is not empty:
   - Dequeue the current module.
   - For each dependency specifier in the module's `dependencies` map (extracted during parsing):
     - Resolve the specifier relative to the current module's directory using `resolve()` (Component 2).
     - If the resolved module ID is not already in the graph:
       - Read, parse, and create a `ModuleNode` for it (with `isEntry: false`).
       - Add the new module to the graph.
       - Enqueue it for further exploration.
     - Link the dependency bidirectionally using `ModuleGraph.linkDependency()`.
4. **Completion**: When the queue empties, the graph contains every module reachable from any entry point—the complete transitive closure.

> **Critical Insight**: This BFS approach naturally handles circular dependencies during construction because we add modules to the graph as we discover them, then later establish the bidirectional links. Circular dependencies don't break the graph structure—they create cycles that must be handled specially during topological sorting.

Stage 2: Mark Used Exports via Static Analysis

With the complete graph built, we now determine which exports are actually used—the foundation for tree shaking.

1. **Start from Entry Points**: Begin the marking process from each entry module. Entry modules are considered to "use" all their named imports and any namespace imports (`import *`). They also implicitly "use" the entire module if they perform a side-effect-only import (`import "./polyfill"`).
2. **Recursive Export Marking**: For each module being processed:
   - For each import statement in the module:
     - **Named Imports**: For `import { foo, bar } from "./module"`, mark `foo` and `bar` as used exports in the dependency module.
     - **Namespace Imports**: For `import * as ns from "./module"`, mark **all** exports of the dependency module as used (since any could be accessed via the namespace object).
     - **Default Imports**: For `import defaultExport from "./module"`, mark the `"default"` export as used.
     - **Side-Effect Imports**: For `import "./module"` (no bindings), don't mark specific exports, but ensure the module is included via side effect detection.
   - For re-exports (`export { foo } from "./module"` or `export * from "./module"`):
     - Treat these as both an import (marking the source module's exports) and an export (adding to current module's exports).
     - For `export *`, mark all exports from the source module.
3. **Propagation**: When an export is marked as used in a module, we must also consider:
   - If this module re-exports those marked exports, we need to propagate the "used" marking to the module's own exports that correspond to those re-exports.
   - This creates a chain reaction: marking an export in module A might cause exports in module B (which re-exports from A) to become marked, which then affects module C that imports from B.

4. **Side Effect Handling**: Modules with `sideEffects: false` in their `package.json` are assumed to be pure—if none of their exports are marked as used, the entire module can be removed. Modules without this field or with `sideEffects: true` are always included if reached, regardless of export usage.

The algorithm uses a worklist approach: when an export is newly marked in a module, we add that module to a worklist to check if its own exports (which might be re-exports) now become used.

Stage 3: Prune Unused Nodes and Exports

After marking, we remove dead code in two passes:

1. **Module Pruning**: Iterate through all modules in the graph. For each module:
   - If `module.shouldIncludeInBundle()` returns `false`, remove the module from the graph.
   - When removing a module, also remove all edges (dependencies/dependents) connecting to it.
2. **Export Pruning**: For remaining modules, filter their `exports` set to only include those marked in `usedExports`. Unused exports are discarded.

> **Performance Consideration**: In practice, we perform pruning in reverse topological order (dependents before dependencies) so that when we remove a module, we've already processed modules that depend on it. This prevents dangling references.

Stage 4: Topological Sort for Bundling

Finally, we determine the correct execution order for modules in the bundle:

1. **Kahn's Algorithm**: Use Kahn's algorithm for topological sorting, which handles cycles gracefully by detecting them.
2. **Cycle Handling**: If a cycle is detected (circular dependencies), we break the cycle by:
   - Grouping the cyclic modules together.
   - Ensuring they're placed consecutively in the output.
   - The runtime module loader (Component 4) must handle execution order within cycles.
3. **Sort Result**: Return modules in **post-order dependency order**, meaning dependencies come before dependents. This ensures when a module's factory function executes, all its dependencies are already loaded.

ADR: Tree Shaking Algorithm Choice

> **Decision: Mark-and-Sweep Static Analysis**
>
> - **Context**: We need to eliminate dead code from the final bundle while preserving all necessary code and side effects. The algorithm must handle complex re-export patterns, circular dependencies, and the `sideEffects` package.json field. It must be understandable for educational purposes while being effective for real-world codebases.
> - **Options Considered**:
>   1. **Mark-and-sweep from entry points**: Start from entry modules, traverse imports, mark used exports, then remove unmarked exports and modules.
>   2. **Reverse import graph traversal**: Build a reverse graph of export→importers, then trace from entry points through exports to find reachable code.
>   3. **Symbol-level reachability**: Track each individual symbol through the graph, following assignments and references.
> - **Decision**: We chose **Option 1: Mark-and-sweep from entry points**.
> - **Rationale**:
>   - **Conceptual clarity**: The algorithm mirrors garbage collection techniques familiar to developers, making it easier to understand and debug.
>   - **Incremental marking**: The worklist-based approach naturally handles complex re-export chains without requiring multiple passes.
>   - **Integration simplicity**: It fits cleanly with our existing graph structure—we can reuse the same `ModuleGraph` and `ModuleNode` representations without major refactoring.
>   - **Adequate precision**: For educational purposes, it provides sufficient dead code elimination while avoiding the complexity of full symbolic execution, which would be overkill for our scope.
> - **Consequences**:
>   - **Positive**: Relatively simple to implement and explain. Handles the majority of tree shaking cases in real codebases.
>   - **Negative**: Less precise than symbol-level analysis for complex patterns like dynamically accessed exports (`imported[someVariable]`). We conservatively mark the entire namespace as used in such cases.
>   - **Maintenance**: The algorithm's simplicity means edge cases must be explicitly handled (e.g., `import *` marks all exports).

Algorithm Comparison Table

Option	Pros	Cons	Why Not Chosen
Mark-and-sweep from entry	Simple, intuitive analogy to GC; Handles re-exports well; Single-pass with worklist	Less precise for dynamic access; Conservative with namespace imports	**CHOSEN** - Best balance of simplicity and effectiveness
Reverse import graph	Efficient for finding unused exports; Good for incremental builds	Complex to implement; Harder to debug; Doesn't handle re-export chains naturally	Too complex for educational context
Symbol-level reachability	Most precise elimination; Handles dynamic patterns better	Very complex; Requires deep AST analysis; Heavy computational cost	Overkill for our scope; would obscure core concepts

Common Pitfalls

Building and transforming the module graph presents several subtle challenges that frequently trap developers:

⚠️ Pitfall 1: Circular Dependencies Breaking Topological Sort

Description: Attempting to topologically sort a graph with cycles using a naive algorithm will either infinite loop or incorrectly report no valid ordering.

Why it's wrong: JavaScript modules with circular dependencies are valid and common. The bundler must handle them correctly, not fail.

Fix: Use Kahn's algorithm with cycle detection. When cycles are detected, group the cyclic modules together and output them consecutively. The runtime module loader must support resolving circular dependencies by allowing partially loaded modules.

⚠️ Pitfall 2: Missing Side Effects in package.json

Description: Assuming all modules without explicit `sideEffects: false` have side effects, causing unnecessary code inclusion. Conversely, incorrectly treating modules with `sideEffects: false` as pure when they actually have side effects.

Why it's wrong: Over-inclusion bloats bundle size; under-inclusion breaks application functionality.

Fix:

1. Always read the `sideEffects` field from a module's `package.json`.
2. If `sideEffects: false`, the module can be fully removed if none of its exports are used.
3. If `sideEffects` is an array, those specific files/modules have side effects even if exports unused.
4. For modules without `package.json` or `sideEffects` field, conservatively assume they have side effects.
5. Perform additional static analysis to detect obvious side effects like top-level function calls, assignments to global variables, or `useEffect`-style calls.

⚠️ Pitfall 3: Incorrectly Removing Code with Global Side Effects

Description: Removing polyfills, CSS-in-JS initialization, or auto-registration code that has global side effects but isn't explicitly imported.

Why it's wrong: The application breaks in production because essential setup code is missing, even though the bundle appears "correct."

Fix:

1. Always include modules that are imported for side effects only (`import "./polyfill"`).
2. Implement basic side effect detection in the parser: flag modules containing:
   - Top-level function calls (except declarations)
   - Assignments to `window`, `global`, `document`, etc.
   - `Object.defineProperty` calls on global objects
3. Mark such modules with `hasSideEffects: true` regardless of export usage.

⚠️ Pitfall 4: Mishandling Re-export Chains

Description: Failing to propagate "used" markings through multiple layers of re-exports, causing elimination of actually used code.

Why it's wrong: Code that's imported via `import { foo } from "./index"` where `index.js` re-exports from `./internal` might incorrectly remove the implementation in `internal.js`.

Fix: Implement proper re-export tracking in the marking algorithm. When marking an export in module A, also check if any module B re-exports that symbol, and mark the corresponding export in B. This requires maintaining a mapping from original export to re-export locations.

⚠️ Pitfall 5: Dynamic Import() Treated as Static Dependency

Description: Adding dynamically imported modules (`import("./module")`) to the main graph and including them unconditionally in the initial bundle.

Why it's wrong: Defeats the purpose of code splitting—dynamic imports should create separate bundles that load on demand.

Fix:

1. During parsing, distinguish between static `import` statements and dynamic `import()` calls.
2. Add dynamic imports to a separate `dynamicDependencies` map in `ModuleNode`.
3. During graph building, process dynamic imports separately—they become entry points for separate bundles (code splitting), not part of the main graph's transitive closure.
4. The marking algorithm should start from both static entry points AND dynamic import locations when marking used exports for each split point.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Graph Data Structure	`Map` and `Set` for adjacency lists	Custom optimized graph library (e.g., `graphology`)
Tree Shaking Algorithm	Mark-and-sweep with worklist	On-demand demand analysis with persistent data structures
Cycle Detection	Kahn's algorithm with Tarjan's SCC fallback	Incremental cycle detection during graph building

B. Recommended File/Module Structure

```
bundler/
├── src/
│   ├── index.ts                    # Main CLI entry point
│   ├── graph/                      # This component
│   │   ├── ModuleGraph.ts          # Main graph class
│   │   ├── ModuleNode.ts           # Module node class
│   │   ├── tree-shake.ts           # Tree shaking algorithms
│   │   ├── topological-sort.ts     # Sorting algorithms
│   │   └── index.ts                # Public exports
│   ├── parser/                     # Component 1
│   ├── resolver/                   # Component 2
│   └── generator/                  # Component 4
└── tests/
    └── graph/
        ├── ModuleGraph.test.ts
        └── tree-shake.test.ts
```

C. Infrastructure Starter Code

Complete ModuleNode Class (ready to use):

TYPESCRIPT

```typescript
// src/graph/ModuleNode.ts

import type { ImportSpecifier, ExportSpecifier } from '../parser/types';


export class ModuleNode {

  // Core identity

  id: string;                    // Unique identifier (resolved absolute path)

  filePath: string;              // Absolute file system path

  source: string;                // Original source code

  ast: any;                      // AST from parser



  // Graph relationships

  dependencies: Map<string, string> = new Map();  // specifier → moduleId

  dependents: Set<string> = new Set();            // moduleIds that depend on this module



  // Import/export information

  imports: Map<string, ImportSpecifier> = new Map();  // localName → import info

  exports: Set<string> = new Set();                   // All export names (including 'default')

  reexports: Map<string, { source: string, exported: string }> = new Map(); // localName → source info



  // Tree shaking state

  usedExports: Set<string> = new Set();           // Exports marked as used

  sideEffects: boolean = true;                    // Whether module has side effects

  isEntry: boolean = false;                       // Is this an entry point?



  // Dynamic imports for code splitting

  dynamicImports: Array<{ specifier: string, location: any }> = [];



  constructor(id: string, filePath: string, source: string, ast: any, isEntry: boolean = false) {

    this.id = id;

    this.filePath = filePath;

    this.source = source;

    this.ast = ast;

    this.isEntry = isEntry;

  }



  addDependency(specifier: string, moduleId: string): void {

    this.dependencies.set(specifier, moduleId);
```

```
  }

  addExport(exportName: string): void {
    this.exports.add(exportName);
  }

  markExportUsed(exportName: string): void {
    this.usedExports.add(exportName);
  }

  shouldIncludeInBundle(): boolean {
    // Always include entry modules
    if (this.isEntry) return true;

    // Include if any exports are used
    if (this.usedExports.size > 0) return true;

    // Include if module has side effects
    if (this.sideEffects) return true;

    // Otherwise, safe to remove
    return false;
  }
}
```

D. Core Logic Skeleton Code

ModuleGraph Class Skeleton:

TYPESCRIPT

```typescript
// src/graph/ModuleGraph.ts

import { ModuleNode } from './ModuleNode';

import { resolve } from '../resolver';

import { parseModule, extractDependencies } from '../parser';

import { readFile } from '../fs-cache';


export class ModuleGraph {

  nodes: Map<string, ModuleNode> = new Map();



  addModule(module: ModuleNode): void {

    if (this.nodes.has(module.id)) {

      throw new Error(`Module with id ${module.id} already exists in graph`);

    }

    this.nodes.set(module.id, module);

  }



  getModule(id: string): ModuleNode | undefined {

    return this.nodes.get(id);

  }



  linkDependency(importerId: string, specifier: string, dependencyId: string): void {

    const importer = this.getModule(importerId);

    const dependency = this.getModule(dependencyId);



    if (!importer || !dependency) {

      throw new Error(`Cannot link non-existent modules: ${importerId} -> ${dependencyId}`);

    }



    // Add to importer's dependencies

    importer.addDependency(specifier, dependencyId);



    // Add to dependency's dependents

    dependency.dependents.add(importerId);

  }



  async build(entryPaths: string[]): Promise<ModuleGraph> {

    // TODO 1: Initialize queue for BFS traversal
```

```
  // TODO 2: For each entry path:
  //   a) Resolve to absolute path using resolve()
  //   b) Read file content using readFile()
  //   c) Parse module using parseModule() to create ModuleNode with isEntry: true
  //   d) Add module to graph via addModule()
  //   e) Enqueue module for processing
  // TODO 3: While queue is not empty:
  //   a) Dequeue current module
  //   b) For each dependency in module.dependencies (from parsing):
  //      i) Resolve specifier relative to current module's directory
  //      ii) If resolved module ID not in graph:
  //          - Read, parse, create ModuleNode (isEntry: false)
  //          - Add to graph
  //          - Enqueue for processing
  //      iii) Link dependency using linkDependency()
  // TODO 4: Return this graph for method chaining
  return this;
}

markUsedExports(entryModuleId: string): void {
  // TODO 1: Initialize worklist with entry module
  // TODO 2: While worklist is not empty:
  //   a) Pop current module from worklist
  //   b: For each import in module.imports:
  //      - Named imports: mark corresponding export in dependency as used
  //      - Namespace imports: mark ALL exports in dependency as used
  //      - Default imports: mark 'default' export in dependency as used
  //      - If dependency has newly marked exports, add it to worklist
  //   c) For each re-export in module.reexports:
  //      - If the source export is marked as used in source module,
  //        mark the corresponding re-export in current module as used
  //      - If current module's export is newly marked, add current module to worklist
  // TODO 3: Handle side-effect-only imports: ensure those modules are included
}

shake(): ModuleGraph {
```

```
    // TODO 1: First pass - remove unused modules
    //   For each module in graph:
    //     - If !module.shouldIncludeInBundle():
    //       a) Remove from this.nodes
    //       b) Remove from all dependents' dependencies
    //       c) Remove from all dependencies' dependents
    // TODO 2: Second pass - filter unused exports
    //   For each remaining module:
    //     - Filter module.exports to only include those in module.usedExports
    //     - Also filter reexports map accordingly
    // TODO 3: Return this graph for method chaining
    return this;
  }

  topologicalSort(): ModuleNode[] {
    // TODO 1: Use Kahn's algorithm for topological sort
    //   a) Calculate in-degree (number of dependencies) for each node
    //   b) Initialize queue with nodes having in-degree 0
    //   c) While queue not empty:
    //      - Remove node from queue, add to sorted list
    //      - For each dependent of node:
    //         * Decrease dependent's in-degree
    //         * If in-degree becomes 0, add to queue
    // TODO 2: If sorted list length < total nodes, graph has cycles
    //   a) Use Tarjan's algorithm to find strongly connected components (SCCs)
    //   b) Group cyclic modules together
    //   c) Return flattened list with SCCs as groups
    // TODO 3: Return sorted array of ModuleNodes
    return [];
  }
}
```

E. Language-Specific Hints

- **TypeScript Benefits**: Use discriminated unions for import/export types to ensure exhaustive checking in tree shaking logic.
- **Map/Set Operations**: Use `Map.prototype.forEach` and `Set.prototype.has` for efficient graph operations. Remember that object references work as keys in Maps/Sets.
- **Async/Await**: The `build()` method is async due to file I/O and resolution. Use `Promise.all()` for parallel processing where possible.
- **Memory Management**: For large graphs, consider weak references for `dependents` sets if modules are removed during shaking.

- **Circular Import Prevention**: Keep the `ModuleNode` class independent of the `ModuleGraph` to avoid circular imports in TypeScript.

F. Milestone Checkpoint

After implementing the Graph Builder & Transformer, verify correctness with:

Command:

```bash
node test-graph.js
```

Expected Behavior:

1. Graph builds from a simple entry point with dependencies
2. Tree shaking removes unused exports
3. Topological sort returns correct order
4. Circular dependencies are handled gracefully

Test Case Structure:

```javascript
// test-graph.js
const { ModuleGraph } = require('./src/graph');
const graph = new ModuleGraph();

// Build graph from fixture
await graph.build(['./fixtures/entry.js']);

// Should contain 3 modules: entry, util, helper
console.assert(graph.nodes.size === 3, 'Should have 3 modules');

// Mark used exports
graph.markUsedExports(entryModuleId);

// Tree shaking
graph.shake();

// Should still have 3 modules (all used)
console.assert(graph.nodes.size === 3, 'All modules should remain');

// Topological sort
const sorted = graph.topologicalSort();
console.assert(sorted[0].id.includes('helper'), 'Helper should come first');
console.assert(sorted[2].id.includes('entry'), 'Entry should come last');
```

Signs of Problems:

- **Infinite loop during build**: Likely circular resolution or missing base case in BFS.
- **Missing modules after shaking**: Check `shouldIncludeInBundle()` logic and side effect detection.
- **Wrong topological order**: Verify Kahn's algorithm implementation, especially in-degree calculation.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Module missing from final bundle	Tree shaking too aggressive	Log `shouldIncludeInBundle()` for each module; check `sideEffects` flag	Ensure side effect detection includes global assignments
Runtime error: "exports is not defined"	Unused export removed but still referenced	Check if re-export chains properly marked	Debug `markUsedExports()` with worklist visualization
Circular dependency infinite loop	Topological sort algorithm doesn't handle cycles	Add cycle detection and SCC grouping	Implement Tarjan's algorithm for strongly connected components
Bundle size larger than expected	Side effects preventing removal	Audit `package.json` `sideEffects` fields; analyze module contents	Improve side effect detection in parser
Wrong execution order	Topological sort incorrect	Print dependency graph and verify Kahn's algorithm steps	Ensure in-degree counts all dependencies, not just direct ones

Component 4: Code Generator & Runtime

> **Milestone(s):** Milestone 3

The **Code Generator & Runtime** is the final, integrative component of our bundler, responsible for assembling the analyzed and transformed module graph into a single, executable JavaScript file. This component faces the critical challenge of preserving correct JavaScript runtime semantics while collapsing dozens of separate module files into a unified bundle. Think of it as the final act of a complex theatrical production—all the rehearsed scripts (modules) must be rewritten into a single playbook with clear instructions for the actors (runtime) to perform them in the correct order.

At this stage, the intelligent decisions made earlier—the resolved dependencies, pruned dead code, and sorted execution order—must be rendered into concrete, runnable code. The generator must craft a custom runtime environment that mimics how native ES modules or CommonJS loaders behave, ensuring that `import` statements resolve to the correct values, that circular dependencies don't cause infinite loops, and that live bindings (the dynamic linking of exported variables in ESM) work as expected. This component bridges the gap between the static analysis world and the dynamic execution world.

Mental Model: The Play Script Rewriter

Imagine you're the director of a sprawling theatrical production with dozens of separate actor scripts (modules). Each script defines characters (exports) and references characters from other scripts (imports). To simplify the performance, you must:

1. **Combine all scripts into one master script (the bundle).**
2. **Rewrite character names to avoid conflicts:** `Hero` from `scriptA` and `Hero` from `scriptB` need distinct internal names so they don't collide.
3. **Add detailed stage directions (the runtime):** Instructions on when each original script's content should be "performed" (executed), in what order, and how to hand characters (exported values) between them.
4. **Preserve the original script's formatting cues (source maps):** So if an actor forgets a line during rehearsal, you can trace it back to the exact page and line of the original, separate script.

The Code Generator performs exactly this transformation. It takes the ordered list of `ModuleNode` objects (the scripts, already sorted so dependencies come before dependents) and rewrites their abstract syntax trees (ASTs). It replaces every `import` statement with a function call to the runtime's lookup system, and every `export` statement with an assignment to the runtime's export registry. Finally, it wraps each module's code in a function wrapper (a "scene") to isolate its variables, and concatenates them all together, prefixed by the runtime's "stage manager" code that knows how to load and connect these scenes.

Interface

The primary interface for this component is a single function that consumes the processed `ModuleGraph` and produces one or more `BundleChunk` outputs. Its signature and the key data structures it manipulates are defined below.

Core Generation Function

Method	Parameters	Returns	Description
`generateBundle(moduleGraph, options)`	`moduleGraph: ModuleGraph` `options: BundlerOptions`	`Promise<BundleChunk[]>`	The main entry point for bundle generation. Takes the final, shaken module graph and bundler configuration, and produces one or more output chunks. For our single-bundle case, returns an array with one `BundleChunk`.

Supporting Data Structures

Type	Fields	Description
`BundleChunk`	`id: string`	A unique identifier for this chunk (e.g., `"main"`).
	`modules: Set<string>`	The set of `ModuleNode.id` values included in this chunk.
	`code: string`	The final generated JavaScript code for this chunk.
	`sourceMap: SourceMapV3 | null`	The Source Map V3 object mapping the bundled code back to original sources.
`SourceMapV3`	`version: number`	Always `3`.
	`file: string`	The name of the generated bundle file this source map is for.
	`sourceRoot: string`	An optional root URL for all `sources` entries. Can be empty string.
	`sources: string[]`	An array of URLs to the original source files.
	`sourcesContent: string[]`	An optional array of the original source file contents.
	`names: string[]`	An array of symbol names used in the source.
	`mappings: string`	The VLQ-encoded string mapping generated lines/columns to source lines/columns.
`BundlerOptions` (Relevant subset)	`outFile: string`	The filename for the primary output bundle.
	`sourceMap: boolean`	Whether to generate source maps.
	`minify: boolean`	Whether to minify the output code (out of scope for our basic implementation).

Code Generation Algorithm

The generation process is a sequential, deterministic pipeline that transforms the module graph into executable code. The following numbered steps describe the algorithm for a single, non-code-split bundle.

1. **Initialize Output and Source Map State:**
    - Create a new `BundleChunk` object with a unique `id` (e.g., `"main"`).
    - Initialize an empty array to hold the generated code segments for each module.
    - If `options.sourceMap` is `true`, initialize a `SourceMapGenerator` (or equivalent data) with the `file` set to `options.outFile`. Prepare arrays to accumulate `sources`, `sourcesContent`, and `mappings`.
2. **Generate Runtime Bootstrapper Code:**
    - Prepend a string of JavaScript code that defines the **module runtime**. This runtime is an immediately invoked function expression (IIFE) that creates a private scope and returns the entry point module's exports. It contains:
        - A `modules` registry: An object or Map that will store module factory functions and their cached exports.
        - A `require` or `__require__` function: This is the runtime's module loader. It takes a module ID, checks the cache, executes the module's factory function if not already loaded (providing it with a localized `require`, `exports`, and `module` object), and returns the

cached `exports`.
    - Logic to handle ES Module **live bindings**: The runtime must ensure that when an exported mutable variable is updated, all importing modules see the updated value. This is typically done by making the `exports` object a collection of getter/setter functions or using a `Proxy`.
  - This bootstrapper code is static—the same for every bundle—and does not require source mapping.
3. **Topologically Sort the Module Graph:**
   - Call `moduleGraph.topologicalSort()` to obtain an array of `ModuleNode` objects where every module appears *after* all of its dependencies. This order is critical for correct execution, as a module cannot be evaluated until the modules it imports have been evaluated and their exports are ready.
   - Handle cycles (circular dependencies) by using an algorithm like Tarjan's or Kosaraju's to detect **strongly connected components** (SCCs). Within an SCC, modules must be loaded in an order that works with the runtime's lazy execution semantics, often requiring special handling in the runtime's `require` function.
4. **For Each Module in Sorted Order, Generate Wrapped Code:**
   - This is the core transformation loop. For each `ModuleNode`: a. **Create a Module Factory Function:** Start generating a string that defines a function. This function will typically accept three parameters: `require`, `exports`, and `module`. Example: `function (require, exports, module) { ... }`. b. **Rewrite the Module's AST:** Traverse the module's AST (stored in `ModuleNode.ast`) and perform the following transformations: * **Import Rewriting:** Replace all static `import` statements (e.g., `import { foo } from './bar'`) with a variable declaration that calls the localized `require` function and destructures the needed export. Example: `const { foo } = require('./bar')`. The specifier (`'./bar'`) must be replaced with the resolved module's numeric or hashed ID known to the runtime. * **Export Rewriting:** Replace all `export` statements. * `export const x = 5;` becomes `exports.x = 5;` (for named exports). * `export default function() {}` becomes `exports.default = function() {};` or `module.exports = function() {};` depending on the target module system emulation. * Re-exports (`export { foo } from './bar'`) are replaced with a dynamic assignment: `exports.foo = require('./bar').foo`. * **Dynamic Import Rewriting:** Replace `import()` expressions with a promise-based call to a special runtime function (e.g., `__dynamicImport__`) that can trigger code splitting. In our single-bundle scenario, this may resolve to the same `require` but wrapped in `Promise.resolve()`. * **CommonJS `require` Preservation:** If the source module is CommonJS, its existing `require` calls are left as-is, but the *outer* `require` parameter passed to the factory function will be our runtime's resolver. This creates a hybrid system. c. **Print Transformed Code:** Use a code printer (like `@babel/generator`) to convert the transformed AST back into a JavaScript string. d. **Apply Source Mapping:** If generating source maps, during the printing step, record the mapping between every line/column in the generated module wrapper code and the corresponding line/column in the *original* source file. Add the original `filePath` and `sourceContent` to the source map's `sources` and `sourcesContent` arrays. e. **Add to Registry:** Append a line to the bundle code that registers this factory function in the runtime's `modules` object, keyed by the module's unique ID. Example: `modules[<moduleId>] = function(require, exports, module) { ... };`.
5. **Invoke the Entry Point:**
   - After all module factory functions are registered, append a final line of code to the bundle that calls the runtime's `require` function with the ID of the entry point module. This triggers the execution of the entire dependency graph. Example: `return require('<entryModuleId>');`.
6. **Finalize Bundle and Source Map:**
   - Concatenate all code segments: 1) Runtime bootstrapper, 2) Module registrations, 3) Entry point invocation.
   - If source maps are enabled, generate the final `mappings` string in VLQ format and serialize the complete `SourceMapV3` object to a JSON string. The bundle code should end with a special comment linking to the source map: `//# sourceMappingURL=<outFile>.map`.
   - Assign the final concatenated code string to `BundleChunk.code` and the source map object to `BundleChunk.sourceMap`.
7. **Return Output:** Return an array containing the completed `BundleChunk`.

ADR: Module Wrapping Strategy

> **Decision: Function Wrapper with Explicit `require`, `exports`, `module` Parameters**
>
> - **Context:** We need to isolate each module's top-level scope to prevent variable collisions and to provide a controlled environment for module loading. The wrapper must also facilitate the emulation of both ESM and CommonJS semantics, including live bindings and circular dependency support.
> - **Options Considered:**
>   1. **Immediately Invoked Function Expression (IIFE) per module:** Each module's code is placed inside an IIFE that is executed immediately during bundle execution. Exports are assigned to a global registry.
>   2. **Function Wrapper stored for later execution:** Each module's code is wrapped in a function *definition* that is stored in a registry. This function is only invoked when the module is first `require`d, receiving `require`, `exports`, and `module` as arguments.
> - **Decision:** We chose **Option 2 (Function Wrapper)**.
> - **Rationale:**
>   - **Lazy Evaluation:** It correctly models Node.js and CommonJS behavior where a module's top-level code only runs when it's first required, not when the bundle loads. This is crucial for side effect management and performance.
>   - **Circular Dependency Handling:** It enables the runtime to handle circular dependencies. Module A can require Module B while B is still being evaluated, because the `exports` object for B already exists (even if partially populated) and can be passed to A.
>   - **Live Binding Simulation:** By passing a mutable `exports` object reference to the factory function, we can later implement live bindings by defining getters on this object or replacing its properties.
>   - **Clarity of Control Flow:** The explicit `require` parameter makes the dependency graph's execution flow more transparent in the generated code, which aids debugging.
> - **Consequences:**
>   - **Increased Runtime Complexity:** The bundle must include a non-trivial runtime manager to store and invoke these factory functions.
>   - **Slight Performance Overhead:** There is an extra function call per module on first require compared to IIFEs which execute immediately. However, this is the standard model and is negligible.
>   - **Faithful Emulation:** This approach allows us to closely mimic the behavior of real module systems, which is a primary educational goal.

Option	Pros	Cons	Chosen?
IIFE per module	Simpler generated code, no runtime loader needed.	Executes all modules immediately, breaking lazy loading semantics. Harder to implement circular dependencies and live bindings.	❌
Function Wrapper	Enables lazy evaluation, proper circular dependency support, clear path to live bindings. Faithfully emulates Node.js/CommonJS.	Requires a runtime loader, slightly more complex generated code.	✅

Common Pitfalls

⚠️ Pitfall: Incorrect Export Rewriting for Default vs Named

- **Description:** Mistakenly rewriting `export default MyComponent` as `exports.default = { default: MyComponent }` (double nesting) or treating a default export as just another named export called `'default'` without adjusting the import side accordingly.
- **Why it's Wrong:** In ES modules, `import foo from './mod'` binds `foo` to the module's *default* export. If the exporting module incorrectly stores its default export, the import will receive `undefined` or the wrong value. This breaks one of the most common ESM patterns.
- **Fix:** Be explicit and consistent. For CommonJS-targeted output, a default export should typically set `module.exports = ...`. For ESM-emulation, you can set `exports.default = ...`, but then the runtime's `require` must know that when a module is imported via a default import, it should return `exports.default` (or the whole `exports` if `exports.__esModule` is true, a Babel interoperability pattern).

⚠️ Pitfall: Breaking Source Map Mappings During Transformation

- **Description:** When rewriting AST nodes (like replacing an `ImportDeclaration` with a `VariableDeclaration`), if you don't preserve the source location information of the original nodes, the source map will map generated code to incorrect or empty locations in the source file.

- **Why it's Wrong:** Debugging becomes impossible. A developer using the bundle cannot set breakpoints in their original source code because the debugger has no accurate mapping.
- **Fix:** When creating new AST nodes during the transformation phase, explicitly copy the `loc` and `start`/`end` properties from the original node you are replacing. Most AST manipulation libraries have helper functions for this (e.g., `t.inherits` in Babel).

⚠️ Pitfall: Misordering Module Execution

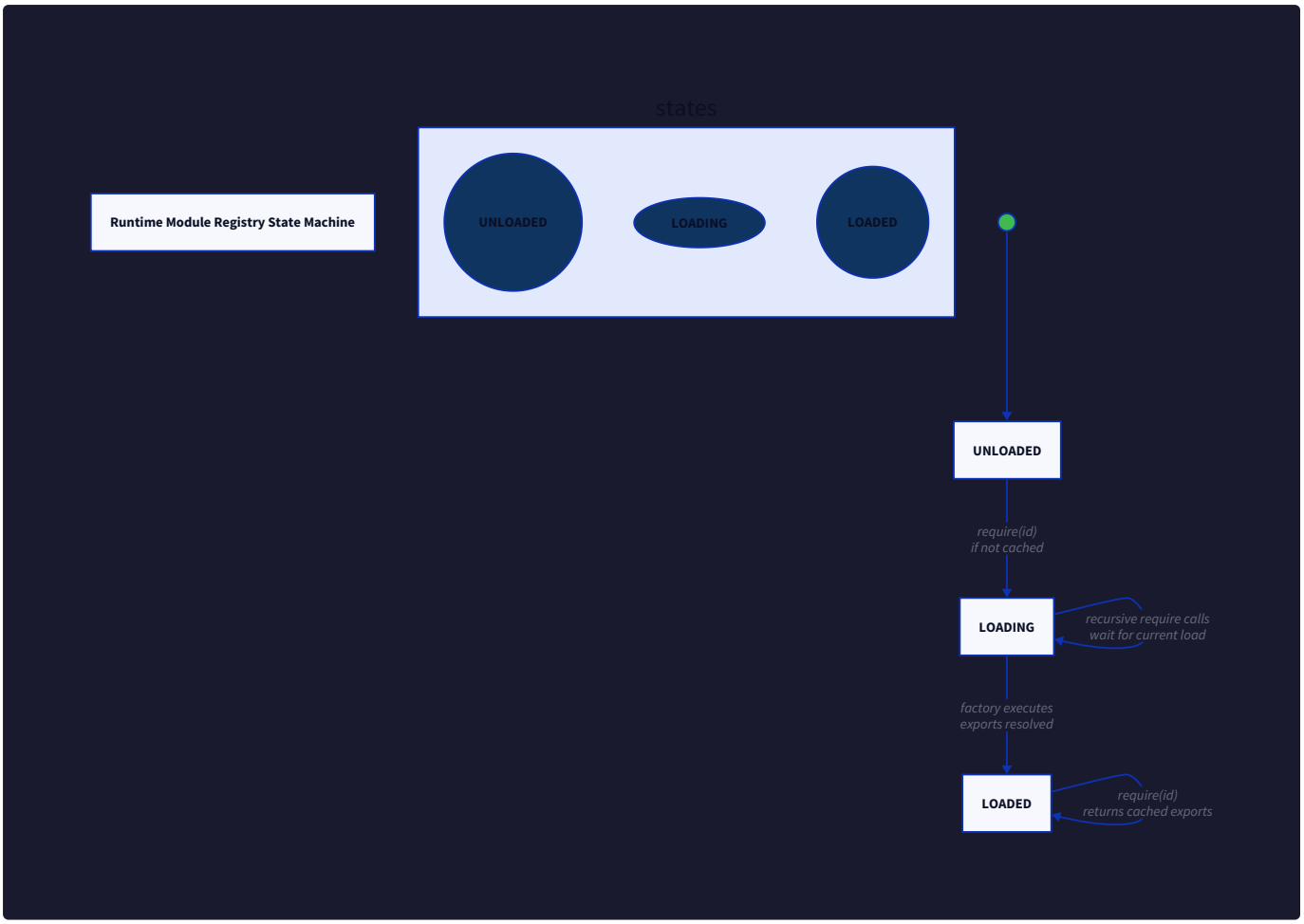
- **Description:** Bundling modules in an order that does not respect the dependency graph (e.g., alphabetical order by file path). A module that `import`s from another may execute before its dependency, leading to `undefined` imports.
- **Why it's Wrong:** JavaScript modules are declarative; the dependencies must be available before the dependent module's top-level code runs. Violating this causes runtime `TypeError`s.
- **Fix:** Always use a **topological sort** on the directed acyclic graph (DAG) of dependencies. Use `ModuleGraph.topologicalSort()`. For cycles, ensure your runtime can handle partially populated `exports` objects (see state machine below).

⚠️ Pitfall: Forgetting to Isolate Module Scope

- **Description:** Simply concatenating module source code without wrapping it, leading to top-level `var`, `let`, `const`, and function declarations from all modules polluting the same shared global scope.
- **Why it's Wrong:** This causes variable name collisions between unrelated modules. A helper function `utils` in module A will overwrite or be overwritten by a `utils` in module B.
- **Fix:** Every module's original code **must** be wrapped inside a function scope (the factory function). Ensure the code printer does not add extra newlines or semicolons outside this wrapper.

Runtime State Machine

The module runtime's internal registry manages the lifecycle of each module. The following state machine describes the lifecycle of a single module from the runtime's perspective. This is a simplified model that our basic runtime can follow.



Current State	Event	Next State	Actions Taken by Runtime
UNLOADED	`require(moduleId)` is called for the first time.	**LOADING**	1. Create a new, empty `exports` object. 2. Create a `module` object: `{ id: moduleId, exports: exports }`. 3. Fetch the module's factory function from the `modules` registry. 4. **Begin execution:** Call the factory function, passing it the scoped `require`, the `exports` object, and the `module` object.
LOADING	The factory function execution **completes successfully**.	**LOADED**	1. The `exports` object (or `module.exports` if it was reassigned) is now populated. 2. Cache this `exports` object in the registry under `moduleId`. 3. Return the cached `exports` to the original caller.
LOADING	During execution, the same `moduleId` is `require`d again (circular dependency).	**LOADING** (Remains)	1. Immediately return the **partially populated** `exports` object to the nested `require` call. 2. This allows the circular dependent to access exports that have been defined up to that point in the factory function's execution.
LOADED	`require(moduleId)` is called again.	**LOADED** (Remains)	1. Immediately return the cached `exports` object from the registry (cache hit). No factory function is executed.

> **Key Insight:** The **LOADING → LOADING** transition for circular dependencies is the critical mechanism that makes them work. The module's code runs only once, but its `exports` object is made available for `require` even mid-execution.

Implementation Guidance

This subsection provides concrete, actionable code to help you implement the Code Generator & Runtime. We'll use JavaScript as the primary language.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
AST Transformation & Printing	`@babel/traverse`, `@babel/generator` (from Babel). Mature, excellent documentation, preserves source map data well.	Directly manipulate the ESTree AST (from Acorn) and write a custom printer. More control but significantly more work.
Source Map Generation	`source-map` library from Mozilla. The industry standard, provides `SourceMapGenerator` class.	Manually encode VLQ mappings. Highly complex and error-prone; not recommended.
Code Concatenation	Template literals and string interpolation. Simple and direct.	Using a string builder or buffer for performance with very large bundles.

B. Recommended File/Module Structure

Integrate the code generator into the existing project structure.

```
build-your-own-bundler/
├── src/
│   ├── bundler.js                  # Main `Bundler.build()` orchestrator
│   ├── index.js                    # CLI entry point
│   ├── parser/
│   │   ├── index.js                # `parseModule`, `extractDependencies`
│   │   └── ... (previous components)
│   ├── resolver/
│   │   └── ... (previous components)
│   ├── graph/
│   │   ├── index.js                # `ModuleGraph` class, `GraphBuilder`
│   │   ├── topological-sort.js     # Kahn's or DFS-based sort
│   │   └── ... (previous components)
│   └── generator/                  # **NEW: This Component**
│       ├── index.js                # `generateBundle` main function
│       ├── runtime-template.js     # String template for the runtime bootstrapper
│       ├── module-rewriter.js      # AST transformation logic
│       ├── source-map-generator.js # Wrapper around `source-map` library
│       └── chunk.js                # `BundleChunk` class definition
├── package.json
└── ... (other config files)
```

C. Infrastructure Starter Code

Here is **complete, ready-to-use** code for a basic `BundleChunk` class and a wrapper for the `source-map` library. The learner can copy these files directly.

File: `src/generator/chunk.js`

JAVASCRIPT

```javascript
/**
 * Represents a single output bundle chunk.
 */
class BundleChunk {
  /**
   * @param {string} id - Unique identifier for the chunk (e.g., 'main').
   * @param {Set<string>} modules - Set of module IDs included in this chunk.
   */
  constructor(id, modules = new Set()) {
    this.id = id;
    this.modules = modules;
    this.code = '';
    this.sourceMap = null;
  }

  /**
   * Adds a module to this chunk.
   * @param {string} moduleId
   */
  addModule(moduleId) {
    this.modules.add(moduleId);
  }

  /**
   * Estimates the size of this chunk in bytes.
   * @returns {number} The length of the `code` string.
   */
  estimateSize() {
    return this.code.length;
  }

  /**
   * Generates a filename for this chunk.
   * Uses the chunk ID or a default.
   * @returns {string}
   */
  getFileName() {
    return `${this.id || 'bundle'}.js`;
```

```
  }
}

module.exports = BundleChunk;
```

File: `src/generator/source-map-generator.js`

JAVASCRIPT

```javascript
const { SourceMapGenerator } = require('source-map');


/**
 * A helper class to manage source map generation.
 * Wraps the `source-map` library for easier use.
 */
class SourceMapManager {
  /**
   * @param {string} file - The name of the generated bundle file.
   * @param {string} sourceRoot - Optional root for all source URLs.
   */
  constructor(file, sourceRoot = '') {
    this.generator = new SourceMapGenerator({ file, sourceRoot });
    this.sourceIndexMap = new Map(); // Maps source path to its index in `sources`
    this.hasContent = false;
  }

  /**
   * Adds a source file to the source map. Must be called before adding mappings for it.
   * @param {string} sourcePath - The path to the original source file.
   * @param {string} sourceContent - The original source code.
   * @returns {number} The index of this source in the `sources` array.
   */
  addSource(sourcePath, sourceContent) {
    if (this.sourceIndexMap.has(sourcePath)) {
      return this.sourceIndexMap.get(sourcePath);
    }
    const index = this.sourceIndexMap.size;
    this.sourceIndexMap.set(sourcePath, index);
    this.generator.setSourceContent(sourcePath, sourceContent);
    this.hasContent = true;
    return index;
  }

  /**
   * Adds a mapping from a generated position to an original source position.
   * @param {Object} mapping
   * @param {Object} mapping.generated - { line, column } in the generated bundle.
```

```
 * @param {Object} mapping.original - { line, column } in the original source.
 * @param {string} mapping.source - The path to the original source file.
 * @param {string} mapping.name - Optional original identifier name.
 */
addMapping(mapping) {
  // Ensure the source is added
  if (!this.sourceIndexMap.has(mapping.source)) {
    throw new Error(`Source "${mapping.source}" not registered. Call addSource first.`);
  }
  this.generator.addMapping(mapping);
}

/**
 * Returns the final Source Map V3 object.
 * @returns {SourceMapV3}
 */
toJSON() {
  return this.generator.toJSON();
}

/**
 * Returns the source map as a base64-encoded string (for inlining).
 * @returns {string}
 */
toBase64() {
  const mapJson = JSON.stringify(this.toJSON());
  return Buffer.from(mapJson).toString('base64');
}

/**
 * Returns the source map as a comment to append to the bundle.
 * @param {string} outFileName - The bundle file name.
 * @param {boolean} inline - If true, inline as base64 data URL.
 * @returns {string} The comment string (e.g., `//# sourceMappingURL=...`)
 */
getComment(outFileName, inline = false) {
  if (inline) {
    const base64 = this.toBase64();
```

```
      return `//# sourceMappingURL=data:application/json;charset=utf-8;base64,${base64}`;
    }
    return `//# sourceMappingURL=${outFileName}.map`;
  }
}

module.exports = SourceMapManager;
```

D. Core Logic Skeleton Code

Now, the core logic that the learner must implement. We provide function signatures and detailed TODO comments that map directly to the algorithm steps.

File: `src/generator/index.js`

JAVASCRIPT

```javascript
const BundleChunk = require('./chunk');

const SourceMapManager = require('./source-map-generator');

const generateRuntimeCode = require('./runtime-template');

const { rewriteModuleAST } = require('./module-rewriter');

const { printAST } = require('../parser/utils'); // Assume a helper that prints AST to string with source map support


/**

 * Generates one or more bundle chunks from the final module graph.

 * @param {ModuleGraph} moduleGraph - The shaken and sorted module graph.

 * @param {BundlerOptions} options - Bundler configuration.

 * @returns {Promise<BundleChunk[]>} Array of output chunks (single chunk for now).

 */

async function generateBundle(moduleGraph, options) {

  // TODO Step 1: Initialize Output and Source Map State

  // 1.1 Create a new BundleChunk (id: 'main').

  // 1.2 Initialize an array `moduleCodeSegments` to hold strings for each module's registration.

  // 1.3 If options.sourceMap is true, create a new SourceMapManager with file: options.outFile.


  // TODO Step 2: Generate Runtime Bootstrapper Code

  // 2.1 Get the runtime code string by calling `generateRuntimeCode()`.

  // 2.2 This code is static and does not need source mapping.


  // TODO Step 3: Topologically Sort the Module Graph

  // 3.1 Call `moduleGraph.topologicalSort()` to get an ordered list of ModuleNodes.

  // 3.2 Store this list in a variable, e.g., `sortedModules`.


  // TODO Step 4: For Each Module in Sorted Order, Generate Wrapped Code

  // 4.1 Begin a loop over `sortedModules`.

  //     For each `moduleNode`:

  //     a. If generating source maps, call `sourceMapManager.addSource(moduleNode.filePath, moduleNode.source)`.

  //     b. Call `rewriteModuleAST(moduleNode, moduleGraph)` to get a transformed AST.

  //     c. Call `printAST(transformedAST, moduleNode.filePath, sourceMapManager)` to get the module's code string and
update source map mappings.

  //     d. Create a module registration line: e.g., `modules[${JSON.stringify(moduleNode.id)}] = function(require, exports,
module) { ${printedCode} };`

  //     e. Push this registration line into `moduleCodeSegments`.


  // TODO Step 5: Invoke the Entry Point

  // 5.1 Determine the entry module ID (the first entry point from options.entryPoints, resolved to its module ID).

  // 5.2 Create an entry invocation line: e.g., `return require(${JSON.stringify(entryModuleId)});`
```

```
  // TODO Step 6: Finalize Bundle and Source Map
  // 6.1 Concatenate all code parts in order: runtime code, moduleCodeSegments joined with newlines, entry invocation line.
  // 6.2 Assign the concatenated string to `chunk.code`.
  // 6.3 If generating source maps:
  //     - Set `chunk.sourceMap = sourceMapManager.toJSON()`.
  //     - Append `sourceMapManager.getComment(options.outFile)` to `chunk.code`.

  // TODO Step 7: Return Output
  // 7.1 Return an array containing the single `chunk`.
}

module.exports = generateBundle;
```

File: `src/generator/module-rewriter.js`

JAVASCRIPT

```javascript
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');

/**
 * Transforms a module's AST to replace imports/exports with runtime calls.
 * @param {ModuleNode} moduleNode - The module to rewrite.
 * @param {ModuleGraph} moduleGraph - The full graph for resolving import IDs.
 * @returns {Object} The transformed AST (a new AST or mutated in-place).
 */
function rewriteModuleAST(moduleNode, moduleGraph) {
  const ast = JSON.parse(JSON.stringify(moduleNode.ast)); // Deep clone to avoid mutating original
  const importReplacements = new Map(); // Store local binding names for imported values

  // TODO: Traverse the AST and perform transformations.
  traverse(ast, {
    // TODO Step 4.b.i: Rewrite Import Statements
    ImportDeclaration(path) {
      // 1. Get the specifier (e.g., './foo').
      // 2. Use moduleGraph to find the dependency's ModuleNode and get its resolved ID.
      // 3. For each specifier in the import statement (ImportSpecifier, ImportDefaultSpecifier, ImportNamespaceSpecifier):
      //    - Determine the local binding name (e.g., `foo` in `import { foo }`).
      //    - Determine the imported name ('default', '*', or a specific name).
      //    - Create a replacement variable declaration: e.g., `const localName = require(moduleId)[importedName]`.
      //    - Record the mapping for potential use in export rewriting.
      // 4. Replace the entire ImportDeclaration node with the new variable declaration(s).
      // 5. Ensure source location info is preserved from the original node.
    },

    // TODO Step 4.b.ii: Rewrite Export Statements
    ExportNamedDeclaration(path) {
      // Handle: export { a, b };
      // Handle: export const c = 1;
      // Handle: export { x } from './other';
      // 1. If it has a `source` (re-export): Replace with `exports.x = require('./other').x`.
      // 2. If it has a declaration: Remove the `export` keyword and add assignments to `exports`.
      //    e.g., `export const c = 1;` -> `const c = 1; exports.c = c;`
      // 3. If it's a list of specifiers without source: Add assignments `exports.a = a;`.
    },
```

```
    ExportDefaultDeclaration(path) {
      // Handle: export default function() {} or export default expression;
      // 1. Remove the `export default` part.
      // 2. Add an assignment: `exports.default = <expression>;` or `module.exports = <expression>;`
    },
    ExportAllDeclaration(path) {
      // Handle: export * from './other';
      // This is complex. For simplicity, we can replace with a loop that copies all exports.
      // e.g., `const dep = require('./other'); for (const key in dep) { if (key !== 'default') exports[key] = dep[key]; }`
    },

    // TODO: Rewrite Dynamic Imports (Bonus)
    CallExpression(path) {
      // if (path.node.callee.type === 'Import') {
      //   Replace `import('./lazy')` with `Promise.resolve(require('./lazy'))` or a special runtime dynamic import function.
      // }
    },

    // TODO: Leave CommonJS `require` calls as-is, they will use the runtime's `require`.
  });

  return ast;
}

module.exports = { rewriteModuleAST };
```

E. Language-Specific Hints

- **AST Cloning:** When manipulating ASTs, it's safer to work on a deep clone (as shown with `JSON.parse(JSON.stringify(...))`) to avoid accidentally mutating the original AST stored in the `ModuleGraph`. In production, use a library like `lodash.clonedeep` for performance.
- **Source Map Positions:** Babel's `@babel/generator` accepts a `sourceFileName` option and can generate source maps automatically if you pass a `sourceMaps: true` option and a `sourceMapTarget`. You can integrate this with the `SourceMapManager` by consuming the raw mappings it produces.
- **Runtime Template:** Store the runtime bootstrapper code as a multi-line template string in a separate file (`runtime-template.js`). Use placeholders like `%ENTRY_MODULE_ID%` if you need to inject values, or keep it static and have the entry point invocation appended after it.

F. Milestone Checkpoint

After implementing the `generateBundle` function and its helpers, you can verify Milestone 3 with the following checkpoint.

Command to Run:

```bash
node src/index.js bundle --entry ./test/fixtures/simple-app/index.js --outDir ./dist
```

Expected Behavior:

1. The command executes without errors.

2. A file `dist/main.js` (or `dist/bundle.js`) is created.
3. The file contains valid JavaScript that can be run in Node.js: `node dist/main.js` should execute the application and produce the expected output (e.g., log messages, computed results).
4. If source maps are enabled (`--sourcemap`), a `.map` file should also be generated, and the bundle should end with a `//# sourceMappingURL=` comment.
5. The bundle should be **self-contained**; running it should not attempt to load external modules via `require('./relative/path')` from the filesystem—all dependencies are included.

Signs of Success:

- No `Cannot find module` errors when executing the bundle.
- Module exports and imports work correctly (values are passed between modules).
- The output code is wrapped in a clear runtime structure, with each module inside a function.

Common Failure Modes and Checks:

- **`SyntaxError` when running the bundle:** The generated JavaScript is invalid. Check the printed code for missing semicolons, unbalanced braces, or incorrect variable declarations. Inspect the output of `printAST`.
- **`undefined` imports:** Modules are likely executing out of order. Verify that `moduleGraph.topologicalSort()` returns a valid order and that you are processing modules in that order.
- **Variables leaking to global scope:** A module's top-level `var` is accessible in another module. Ensure every module's original code is wrapped inside the factory function and that the factory function's body is enclosed in `{ }`.
- **Source map comment points to wrong file:** Check that the `file` property in `SourceMapV3` and the `sourceMappingURL` comment match your actual output file name.

Interactions and Data Flow

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section describes the **runtime orchestration** of our bundler—the precise sequence of operations that transforms a simple command-line invocation into a production-ready bundle. Understanding this flow is crucial for debugging and extending the system. We'll examine both the happy path (successful bundle generation) and the internal communication mechanisms that coordinate the components.

Bundling Sequence

Mental Model: The Assembly Line Factory Imagine our bundler as a specialized factory assembly line. Raw materials (source code files) enter at one end, pass through a series of precisely ordered processing stations (components), and emerge as packaged products (bundles) at the other end. Each station performs a specific transformation, with strict handoff protocols and quality checks between stages. The entire line is controlled by a central production manager (the `Bundler` class) that ensures proper sequencing, error handling, and resource cleanup.

The bundling process follows a **strict unidirectional pipeline pattern**: data flows sequentially through five primary processing stages, with each stage consuming the output of the previous stage and producing input for the next. This design minimizes component coupling and enables clear separation of concerns.

Step-by-Step Pipeline Execution

The following numbered sequence details the complete transformation from CLI invocation to bundle file writing. Each step corresponds to a method call or component interaction in the implementation.

1. **CLI Invocation and Configuration Loading**
   - The user executes the bundler via command line (e.g., `node bundler.js --entry ./src/index.js --out ./dist/bundle.js`).
   - The CLI layer parses arguments and configuration files, constructing a `BundlerOptions` object with validated entry points, output paths, and feature flags.
   - The CLI instantiates the main `Bundler` class, passing the `BundlerOptions` configuration.
2. **Entry Point Discovery and Initial Module Creation**
   - The `Bundler.build()` method initiates the pipeline by calling `GraphBuilder.build(entryPaths)`.

- For each entry path in `entryPoints`, the system creates an initial `ModuleNode` with `isEntry: true`.
- Each entry module undergoes immediate parsing via `parseModule(sourceCode, filePath)` to extract its immediate dependencies.

3. **Graph Construction via Breadth-First Resolution**
   - The graph builder maintains a **worklist algorithm** queue of unresolved module specifiers.
   - For each unresolved specifier in a module's `dependencies` map, the resolver component executes `resolve(specifier, fromDir)`.
   - The resolver performs the complete Node.js resolution algorithm (detailed in Component 2), consulting the `FSCache` for file system reads and maintaining a `ResolutionContext` for symlink tracking and caching.
   - 

   - When resolution succeeds, a new `ModuleNode` is created for the resolved path (unless already in the graph). The system calls `ModuleGraph.linkDependency(importerId, specifier, dependencyId)` to establish **bidirectional navigation** links.
   - The newly discovered module is parsed, its dependencies extracted, and those specifiers are added to the worklist.
   - This process continues until the worklist is empty, yielding the complete **transitive closure** of all reachable modules.

4. **Static Analysis and Tree Shaking**
   - With the complete `ModuleGraph` built, the system performs **mark-and-sweep** dead code elimination.
   - Starting from each entry module, `ModuleGraph.markUsedExports(entryModuleId)` traverses the graph, following import relationships and marking each encountered export as used via `ModuleNode.markExportUsed(exportName)`.
   - The algorithm respects re-export chains by following `reexports` mappings and handles namespace imports (`import * as foo`) conservatively (marking all exports of the referenced module).
   - After marking completes, `ModuleGraph.shake()` removes:
     - Entire modules where `shouldIncludeInBundle()` returns `false` (no used exports and `sideEffects: false`)
     - Individual export statements from modules that have other used exports
   - The resulting pruned graph contains only **reachable code** and essential side effects.

5. **Topological Sorting and Cycle Handling**
   - The system calls `ModuleGraph.topologicalSort()` to determine the correct execution order.
   - The algorithm detects **strongly connected components** (circular dependencies) and collapses each cycle into a single group that will be loaded together.
   - The output is a linear order of modules (or cycle groups) where dependencies appear before dependents (**post-order dependency order**).

6. **AST Transformation and Code Generation**
   - The `generateBundle(moduleGraph, options)` method orchestrates the final code generation.
   - For each module in the topological order:
     - `rewriteModuleAST(moduleNode, moduleGraph)` transforms the module's AST:
       - Replaces `import` statements with calls to the runtime module loader
       - Replaces `export` statements with assignments to the module's exports object

        - Updates `require()` calls for CommonJS interoperability
      - The transformed AST is converted to JavaScript code, wrapped in a **factory function** that provides module isolation.
    - All wrapped module functions are concatenated in dependency order.
    - The runtime loader boilerplate (which manages the module registry and **live bindings**) is prepended to the bundle.
7. **Source Map Generation**
    - During code generation, a `SourceMapManager` tracks every position in the generated bundle.
    - For each original source file, `SourceMapManager.addSource(sourcePath, sourceContent)` registers the original content.
    - As each transformed module's code is generated, `SourceMapManager.addMapping(mapping)` records mappings from generated positions back to original file, line, and column.
    - After all code is generated, `SourceMapManager.toJSON()` produces a complete `SourceMapV3` object.
8. **Output Writing and Finalization**
    - The system creates `BundleChunk` objects containing the generated code and source map.
    - For each chunk, `BundleChunk.getFileName()` generates an output filename based on chunk content hash (for code splitting scenarios).
    - The bundle code is written to the filesystem at the specified `outDir` or `outFile`.
    - If source maps are enabled, they are either embedded as a data URL comment or written as separate `.map` files.
    - The CLI outputs a summary report showing bundle size, module count, and tree-shaking savings.

Data Transformation Through the Pipeline

The following table illustrates how data evolves through each stage of the pipeline:

Pipeline Stage	Input Data	Output Data	Key Transformation
Parsing	Raw source code as strings	`ModuleNode` with AST and dependency list	Text → Structured tree (AST) with extracted relationships
Resolution	Module specifiers + importer directory	Absolute file system paths	Abstract references → Concrete file locations
Graph Building	Collection of `ModuleNode` objects	Complete `ModuleGraph` with bidirectional edges	Isolated modules → Connected dependency graph
Tree Shaking	Complete `ModuleGraph`	Pruned `ModuleGraph` with `usedExports` marked	Full graph → Minimal reachable subgraph
Code Generation	Pruned `ModuleGraph` + topological order	`BundleChunk` with generated code and source map	Abstract graph → Executable JavaScript with runtime
Output	`BundleChunk` objects	Files on disk (.js, .map)	In-memory representation → Persistent artifacts

Concrete Walk-Through Example

Consider a simple project with two files:

- `src/index.js`: `import { add } from './math.js'; console.log(add(2, 3));`
- `src/math.js`: `export const add = (a, b) => a + b; export const multiply = (a, b) => a * b;`

1. The CLI receives `--entry ./src/index.js`.
2. `parseModule` reads `index.js`, produces an AST, and extracts one dependency: `'./math.js'`.
3. `resolve('./math.js', '/project/src')` returns `/project/src/math.js`.
4. `parseModule` reads `math.js`, extracts two exports (`add`, `multiply`).
5. Graph building completes with 2 linked `ModuleNode` objects.
6. `markUsedExports` starts at `index.js`, follows the import of `add`, marks only the `add` export as used in `math.js`.
7. `shake()` removes the `multiply` export from `math.js` (but keeps the module since it has side effects).
8. Topological sort returns: [`math.js`, `index.js`] (dependency before dependent).
9. Code generation rewrites:

- `math.js` : Wraps code, replaces `export const add = ...` with `exports.add = ...`
- `index.js` : Wraps code, replaces `import { add } from './math.js'` with `const { add } = require__('./math.js')`

10. Runtime loader prepended, bundle written to disk. The final bundle contains only the `add` function, not `multiply`.

Internal Message Formats

Mental Model: The Air Traffic Control System Internal communication within the bundler resembles an air traffic control system: different components (planes) broadcast structured messages (flight data) to a central coordinator (control tower) and to each other. These messages follow strict formats (flight plans) and include metadata about their origin, severity, and payload. The system categorizes messages by type (errors, warnings, info) and routes them appropriately—critical errors halt all operations (ground all flights), while warnings allow continued processing with alerts.

The bundler uses three primary message categories: **errors** (unrecoverable conditions), **warnings** (recoverable but notable conditions), and **info logs** (progress diagnostics). All messages flow through a centralized logging system that can filter, format, and output them to console, files, or external monitoring systems.

Error Message Format

Error messages represent unrecoverable failures that halt the bundling process. They follow a consistent structure:

Field	Type	Description
`code`	string	Machine-readable error code (e.g., `MODULE_NOT_FOUND`, `PARSE_ERROR`)
`message`	string	Human-readable description of the error
`severity`	enum	Always `'error'`
`location`	object or null	Source location where error occurred
`location.file`	string	Absolute file path
`location.line`	number	1-based line number
`location.column`	number	0-based column number
`cause`	Error or null	Underlying exception that triggered this error
`specifier`	string or null	Module specifier involved (for resolution errors)
`importer`	string or null	Module importing the problematic specifier
`suggestions`	string[]	Optional fix suggestions

Example Error Scenario: When `resolve('./missing.js', '/src')` fails to find the file, it throws an error with:

```json
{
  "code": "MODULE_NOT_FOUND",
  "message": "Cannot find module './missing.js'",
  "severity": "error",
  "location": { "file": "/src/index.js", "line": 5, "column": 20 },
  "specifier": "./missing.js",
  "importer": "/src/index.js",
  "suggestions": ["Check file exists", "Verify file extension (.js/.ts)"]
}
```

Warning Message Format

Warnings indicate non-fatal issues that don't halt bundling but may affect correctness or performance:

Field	Type	Description
`code`	string	Machine-readable warning code (e.g., `UNUSED_EXPORT`, `CIRCULAR_DEPENDENCY`)
`message`	string	Human-readable warning description
`severity`	enum	Always `'warning'`
`location`	object or null	Source location of concern
`moduleId`	string	ID of module where warning applies
`exportName`	string or null	Specific export involved (for tree shaking warnings)
`recovery`	string	How the system handled the issue

Example Warning Scenario: When tree shaking detects an unused export, it emits:

```json
{
  "code": "UNUSED_EXPORT",
  "message": "Export 'multiply' is never imported",
  "severity": "warning",
  "location": { "file": "/src/math.js", "line": 3, "column": 16 },
  "moduleId": "math.js",
  "exportName": "multiply",
  "recovery": "Export removed from final bundle"
}
```

Log Event Format

Info logs provide diagnostic information about bundler progress and decisions:

Field	Type	Description
`level`	enum	`'debug'`, `'info'`, or `'verbose'`
`stage`	string	Pipeline stage emitting the log
`message`	string	Human-readable log message
`timestamp`	number	Unix timestamp with milliseconds
`data`	object	Structured data relevant to the event
`durationMs`	number or null	Operation duration (for completion events)

Example Log Scenario: After graph building completes, the system logs:

```json
{
  "level": "info",
  "stage": "graph",
  "message": "Module graph built",
  "timestamp": 1678901234567,
  "data": { "moduleCount": 42, "entryPoints": ["./src/index.js"] },
  "durationMs": 125
}
```

Message Routing and Handling

The following state table describes how different components handle incoming messages:

Component	Error Handling	Warning Handling	Log Processing
Parser	Throws `PARSE_ERROR` on syntax errors	Warns on deprecated syntax	Logs file parse times
Resolver	Throws `MODULE_NOT_FOUND` on unresolved imports	Warns on ambiguous `package.json` fields	Logs resolution cache hits/misses
Graph Builder	Throws `CIRCULAR_DEPENDENCY` on unresolvable cycles	Warns on unused exports during marking	Logs graph traversal metrics
Code Generator	Throws `TRANSFORM_ERROR` on AST rewrite failures	Warns on source map generation issues	Logs bundle size estimates
CLI	Prints formatted error and exits with code 1	Prints formatted warning to stderr	Prints summary to stdout based on verbosity

> **Key Design Insight:** The strict message format standardization enables powerful debugging and integration capabilities. External tools can parse these structured messages to provide IDE integration, build dashboard visualizations, or automated fixing suggestions. This transforms the bundler from a black box into an observable system.

Common Pitfalls in Message Handling

⚠️ Pitfall: Silent Error Swallowing

- **Description:** Catching an exception and not re-throwing it as a structured error, causing the bundler to continue with invalid state.
- **Why It's Wrong:** The pipeline assumes each stage's output is valid. Continuing with corrupted data leads to cryptic downstream failures or incorrect bundles.
- **Fix:** Always wrap lower-level exceptions in the appropriate structured error type and propagate immediately.

⚠️ Pitfall: Overly Verbose Logging

- **Description:** Emitting log events for every minor operation (e.g., each file read, each AST node visited).
- **Why It's Wrong:** Performance degrades due to console I/O, and important signals drown in noise.
- **Fix:** Use log levels appropriately: `debug` for internal operations, `info` for stage boundaries, `verbose` only for detailed diagnostics when explicitly enabled.

⚠️ Pitfall: Inconsistent Location Reporting

- **Description:** Reporting error locations with different coordinate systems (0-based vs 1-based line numbers).
- **Why It's Wrong:** External tools consuming these messages cannot reliably map errors to editor positions.
- **Fix:** Standardize on 1-based lines and 0-based columns (matching most editors and the Source Map specification).

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Message Formatting	Plain console.log with prefix strings	Structured JSON logging with `pino` or `winston` libraries
Error Propagation	Try/catch with manual error object creation	Custom Error subclasses with serialization methods
Progress Reporting	Simple console updates	Interactive progress bars with `ora` or `cli-progress`

Recommended File/Module Structure

```
bundler/
  src/
    index.js                        # CLI entry point
    bundler/                        # Main bundler class
      index.js                      # Bundler.build() entry
      messages/                     # Message handling subsystem
        error-codes.js              # Error code constants
        structured-error.js         # StructuredError class
        logger.js                   # Centralized logger
        formatters/                 # Message formatters
          console-formatter.js      # CLI output formatting
          json-formatter.js         # Machine-readable formatting
      pipeline/                     # Pipeline orchestration
        pipeline.js                 # Pipeline runner
        stages/                     # Individual pipeline stages
          parse-stage.js            # Stage 1: Parsing
          resolve-stage.js          # Stage 2: Resolution
          graph-stage.js            # Stage 3: Graph building
          transform-stage.js        # Stage 4: Transformation
          generate-stage.js         # Stage 5: Code generation
      # ... other components (parser, resolver, etc.)
```

Infrastructure Starter Code

Complete Structured Error Class:

JAVASCRIPT

```javascript
// src/bundler/messages/structured-error.js

export class StructuredError extends Error {

  constructor(code, message, options = {}) {

    super(message);

    this.name = 'StructuredError';

    this.code = code;

    this.severity = options.severity || 'error';

    this.location = options.location || null;

    this.specifier = options.specifier || null;

    this.importer = options.importer || null;

    this.suggestions = options.suggestions || [];

    this.cause = options.cause || null;



    // Capture stack trace

    if (Error.captureStackTrace) {

      Error.captureStackTrace(this, StructuredError);

    }

  }



  toJSON() {

    return {

      code: this.code,

      message: this.message,

      severity: this.severity,

      location: this.location,

      specifier: this.specifier,

      importer: this.importer,

      suggestions: this.suggestions,

      stack: this.stack

    };

  }



  toString() {

    let str = `[${this.code}] ${this.message}`;

    if (this.location) {

      str += ` at ${this.location.file}:${this.location.line}:${this.location.column}`;
```

```
    }
    if (this.specifier) {
      str += ` (specifier: ${this.specifier})`;
    }
    return str;
  }
}

// Error code constants
export const ERROR_CODES = {
  MODULE_NOT_FOUND: 'MODULE_NOT_FOUND',
  PARSE_ERROR: 'PARSE_ERROR',
  CIRCULAR_DEPENDENCY: 'CIRCULAR_DEPENDENCY',
  TRANSFORM_ERROR: 'TRANSFORM_ERROR',
  // ... other error codes
};
```

Centralized Logger with Levels:

```javascript
// src/bundler/messages/logger.js

export class Logger {

  constructor(level = 'info', formatter = 'console') {

    this.level = level;

    this.formatter = this.createFormatter(formatter);

    this.levels = { error: 0, warn: 1, info: 2, debug: 3, verbose: 4 };

  }



  createFormatter(type) {

    if (type === 'json') return (log) => JSON.stringify(log);

    return (log) => `[${log.level.toUpperCase()}] ${log.message}`;

  }



  log(level, message, data = {}) {

    if (this.levels[level] > this.levels[this.level]) return;



    const logEntry = {

      level,

      message,

      timestamp: Date.now(),

      ...data

    };



    const output = this.formatter(logEntry);

    const stream = level === 'error' || level === 'warn' ? process.stderr : process.stdout;

    stream.write(output + '\n');

  }



  error(message, data) { this.log('error', message, data); }

  warn(message, data) { this.log('warn', message, data); }

  info(message, data) { this.log('info', message, data); }

  debug(message, data) { this.log('debug', message, data); }

  verbose(message, data) { this.log('verbose', message, data); }

}
```

JAVASCRIPT

Core Logic Skeleton Code

Pipeline Runner Implementation:

JAVASCRIPT

```javascript
// src/bundler/pipeline/pipeline.js

import { Logger } from '../messages/logger.js';

import { StructuredError, ERROR_CODES } from '../messages/structured-error.js';


export class BundlingPipeline {

  constructor(stages, options = {}) {

    this.stages = stages; // Array of stage objects with execute() method

    this.logger = options.logger || new Logger();

    this.context = {}; // Shared context between stages

  }



  async run(initialInput) {

    this.logger.info('Bundling pipeline started');



    let currentData = initialInput;



    // TODO 1: Iterate through each stage in this.stages

    // TODO 2: For each stage, log start with stage name

    // TODO 3: Call stage.execute(currentData, this.context) and capture result

    // TODO 4: Handle stage errors by wrapping in StructuredError with stage context

    // TODO 5: Update currentData with stage result for next stage

    // TODO 6: Log stage completion with duration

    // TODO 7: After all stages complete, log pipeline success and return final data

    // TODO 8: Implement cancellation checkpoints between stages

    // TODO 9: Add timeout handling for each stage execution

    // TODO 10: Clean up any temporary resources in context after completion

  }

}
```

Pipeline Stage Base Class:

JAVASCRIPT

```javascript
// src/bundler/pipeline/stages/base-stage.js

export class PipelineStage {

  constructor(name, logger) {

    this.name = name;

    this.logger = logger;

  }



  async execute(input, context) {

    const startTime = Date.now();

    this.logger.debug(`Stage ${this.name} starting`);



    try {

      // TODO 1: Validate input format specific to this stage

      // TODO 2: Execute stage-specific transformation logic

      // TODO 3: Update context with any stage-specific metadata

      // TODO 4: Return transformed output for next stage

      const output = await this.process(input, context);



      const duration = Date.now() - startTime;

      this.logger.debug(`Stage ${this.name} completed in ${duration}ms`);

      return output;

    } catch (error) {

      const duration = Date.now() - startTime;

      this.logger.error(`Stage ${this.name} failed after ${duration}ms`, { error });



      // Enhance error with stage context

      if (!(error instanceof StructuredError)) {

        throw new StructuredError(

          'PIPELINE_STAGE_ERROR',

          `Stage ${this.name} failed: ${error.message}`,

          { cause: error, location: context.currentLocation }

        );

      }

      throw error;

    }

  }
```

```
  async process(input, context) {
    throw new Error('Subclasses must implement process()');
  }
}
```

Language-Specific Hints

1. **Error Stack Traces:** Use `Error.captureStackTrace()` in Node.js to create clean stack traces that exclude internal framework code.
2. **Async Error Handling:** Ensure all async operations in the pipeline are wrapped in try/catch blocks, as unhandled promise rejections will crash the process.
3. **Streaming Logs:** For large projects, consider streaming logs to a file instead of holding all messages in memory.
4. **Performance Monitoring:** Use `performance.now()` for high-resolution timing of critical operations.
5. **Signal Handling:** In the CLI, handle `SIGINT` (Ctrl+C) to gracefully shutdown the pipeline and clean up temporary files.

Milestone Checkpoint

Checkpoint: End-to-End Integration Test

1. **Command:** Create a test script that runs the full bundler on a sample project:

```bash
node src/index.js --entry ./test-fixtures/simple-app/index.js --out ./dist/bundle.js
```

2. **Expected Output:**
   - Console shows pipeline stage progression messages
   - Final output: "Bundle generated successfully: 2 modules, 15.2 KB"
   - File `dist/bundle.js` contains valid, runnable JavaScript
   - File `dist/bundle.js.map` exists if source maps enabled
3. **Verification Steps:**
   - Run the generated bundle with Node.js: `node dist/bundle.js`
   - Expected: No errors, correct program output
   - Inspect bundle content: should contain wrapped modules and runtime loader
4. **Failure Indicators:**
   - Pipeline halts with structured error message → Check error details
   - Bundle file missing or empty → Check file system permissions
   - Bundle runs but produces wrong output → Check AST transformation logic

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Pipeline hangs indefinitely	Infinite loop in resolution or circular dependency detection	Add debug logging to worklist algorithm; check `visitedSymlinks` set	Implement cycle detection with depth limits; add timeout to pipeline stages
Error messages lack location info	Location not being captured during parsing/transformation	Check if `sourceLocation` is being extracted from AST nodes	Ensure all error-throwing code receives `location` from AST node or import specifier
Bundle runs but imports are undefined	Import/export rewriting produced incorrect runtime calls	Inspect generated code for a single module; compare import statements to rewritten calls	Verify `rewriteModuleAST` replaces imports with correct runtime function signatures
Source maps don't align with original	Mappings generated with wrong coordinates	Use `source-map` library to validate mappings; check line/column offsets	Ensure AST transformation preserves original locations; verify source map manager uses correct source indices

Error Handling and Edge Cases

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

Robust error handling is the difference between a frustrating black box and a reliable development tool. In a bundler, which processes complex dependency graphs across thousands of files, errors are inevitable—from simple typos in import paths to deep semantic issues like circular dependencies that break runtime execution. This section defines how our system detects, categorizes, and responds to these failures. The core philosophy is **fail-fast with clarity**: when the bundler encounters a problem it cannot safely recover from, it should halt immediately and present the developer with a precise, actionable error message. For less severe issues that don't prevent bundle generation, it provides configurable warnings. This approach ensures developers aren't left debugging a cryptic runtime error in a minified bundle, but instead receive feedback at build time where problems are easiest to fix.

Mental Model: The Building Inspector

Imagine the bundler as a **building inspector** for a skyscraper built from modular components (your code). The inspector's job is to verify the structural integrity of the entire building before anyone moves in. Some issues are **critical failures**—like a missing load-bearing wall (an unresolved module)—that require immediate work stoppage. Other issues are **code violations**—like a slightly undersized electrical wire (an unused import that increases bundle size)—that generate warnings in the inspection report but don't prevent occupancy. The inspector doesn't guess or silently patch problems; they document each issue with a precise location, a violation code, and a recommended fix. Our error handling system embodies this inspector: methodical, unambiguous, and focused on preventing runtime catastrophes.

Failure Modes

A bundler operates across multiple phases, each with distinct failure points. The following table catalogs the primary failure modes, their triggers, detection mechanisms, and impact on the bundling process.

Failure Mode	Trigger Event	Detection Point	Impact on Bundle	Example Scenario
Unresolved Module	A module `specifier` cannot be mapped to a filesystem path.	`resolve(specifier, fromDir)` returns `null` or throws.	**Critical.** The dependency graph is incomplete; the generated bundle would throw "Cannot find module" at runtime.	`import { capitalize } from './utils';` but `./utils.js` does not exist.
Parse Error	Source code contains syntax invalid for the targeted ECMAScript version or contains TypeScript syntax without proper configuration.	`parseModule(sourceCode, filePath)` throws during AST generation.	**Critical.** The module cannot be analyzed for dependencies or exports; tree shaking and code generation cannot proceed.	A missing closing brace `}` or an unsupported JSX syntax in a `.js` file.
Circular Dependency	Two or more modules import each other directly or transitively, creating a cycle in the `ModuleGraph`.	`ModuleGraph.topologicalSort()` detects a strongly connected component with more than one node.	**Conditional.** Simple cycles can be handled by the runtime with proper initialization order. Complex cycles with **live bindings** may cause runtime `ReferenceError` if not handled correctly.	`a.js`: `import { b } from './b';` ↔ `b.js`: `import { a } from './a';`
Missing `package.json`	During resolution of a bare specifier, a `package.json` file is expected in a `node_modules` directory but is absent or unreadable.	`_findPackageDir()` locates a directory but `fs.readFile` for `package.json` fails.	**Critical.** The bundler cannot determine the package's entry point or side effect flags.	A corrupted or manually created `node_modules/lodash` directory without its `package.json`.
Invalid `package.json`	A `package.json` file exists but is malformed (invalid JSON) or contains fields with unexpected types.	`JSON.parse()` fails, or validation of `main`, `exports`, `sideEffects` fields fails.	**Critical.** The bundler cannot reliably interpret the package's configuration.	`package.json` with a trailing comma, or `"main": 123` (not a string).
File System Error	Insufficient permissions, disk full, or the file disappears between resolution and reading.	`FSCache.readFile(filePath)` throws an `ENOENT`, `EACCES`, or other system error.	**Critical.** The module source cannot be retrieved for parsing or inclusion.	A module file is deleted after the resolution phase but before the parsing phase.
Symlink Loop	Symbolic links create an infinite cycle during module resolution	`ResolutionContext.visitedSymlinks` detects the same realpath visited more than once.	**Critical.** Would cause infinite recursion and stack overflow in the resolver.	A package symlinks to itself in its `node_modules` during development.

Failure Mode	Trigger Event	Detection Point	Impact on Bundle	Example Scenario
	(e.g., `a → b → a`).			
Unsupported Module Syntax	The source uses a module feature the bundler explicitly does not support (e.g., certain `import.meta` properties).	`extractDependencies(ast)` encounters an AST node it cannot process.	**Configurable.** Can be a warning (if we can safely ignore) or an error.	Using `import.meta.url` for asset loading without a plugin.
Side Effect Detection Error	Code with important side effects (e.g., polyfill installation) is incorrectly removed because the module was marked `"sideEffects": false`.	Manual testing reveals broken runtime behavior.	**Silent but severe.** The bundle is smaller but functionally broken. Hard to detect automatically.	A CSS-in-JS library that registers styles in a top-level call, but its package.json declares no side effects.
Source Map Generation Failure	A mapping from generated code to original source cannot be calculated (e.g., invalid position data).	`SourceMapManager.addMapping()` receives out-of-bounds line/column numbers.	**Non-critical.** The bundle is still valid, but debugging is harder. The bundler should fall back to no source map or a partial one.	A bug in the AST rewriter generates code positions that don't exist in the original source.
Export/Import Binding Mismatch	An import statement references a named export that does not exist in the target module.	During `ModuleGraph.markUsedExports()`, an import's `imported` name is not in the dependency's `exports` Set.	**Configurable.** Can be a warning (like TypeScript) or a strict error.	`import { nonExistent } from './module';` where `./module` only exports `default`.
Dynamic Import Specifier Not Statically Analyzable	A `dynamic import()` uses a template literal or variable, making it impossible to analyze for code splitting at build time.	`extractDependencies` finds a `DynamicImport` with `isStaticSpecifier: false`.	**Non-critical.** The dynamic import will work at runtime but cannot be pre-processed (e.g., for prefetching). Warn the user.	`import( ./locale/${language}.js )`.

Recovery Strategies

Not all failures are equal, and the bundler's response must be tailored to the severity and recoverability of the issue. Our strategy is built on three pillars: **Fail-Fast for Critical Errors**, **Configurable Warnings for Code Quality**, and **Structured Error Reporting** for developer productivity.

Decision: Structured Error Objects Over Throw Strings

- **Context**: The bundler pipeline has many components that can fail. We need a consistent way to capture, enrich, and communicate errors to the end user.
- **Options Considered**:
  1. **Throw plain strings or Error objects**: Simple but lacks structure for machine-readable error codes, locations, or suggestions.
  2. **Use a custom `StructuredError` class**: Encapsulates error metadata (code, location, suggestions) in a consistent format that can be formatted for CLI or IDE integration.
  3. **Use a third-party error diagnostic library**: (e.g., VSCode's `Diagnostic` type) – heavy weight and may not match our needs.
- **Decision**: Implement a `StructuredError` class as defined in the naming conventions.
- **Rationale**: A structured format allows the CLI to consistently output errors in different formats (human-readable, JSON). It enables future integration with IDEs for click-to-navigate error messages. The `suggestions` field is crucial for helping learners understand and fix common mistakes.
- **Consequences**: All components in the pipeline must catch lower-level errors and wrap them in `StructuredError`. This adds some boilerplate but vastly improves the user experience.

Strategy	Applicable Failure Modes	Mechanism	User Experience
Fail-Fast with Descriptive Error	Unresolved Module, Parse Error, Missing `package.json`, File System Error, Symlink Loop.	The pipeline component throws a `StructuredError`. The top-level `Bundler.build()` catches it, prints the formatted error, and exits the process with a non-zero code.	Execution stops immediately. The error message is printed to stderr with syntax highlighting (file path, line/column), an error code (`MODULE_NOT_FOUND`), and potential fixes.
Warning for Non-Critical Issues	Unused import/export, dynamic import with variable specifier, export/import binding mismatch (if not strict).	The component emits a warning object (similar to `StructuredError` but with `severity: 'warning'`). Warnings are collected in a `DiagnosticBag` and printed to stderr after a successful build, or before an error failure.	The build completes successfully. Warnings are aggregated and displayed at the end, often with a count. The user can review and address code quality issues at their leisure.
Configurable Bail-Out Options	Export/import binding mismatch, certain circular dependency patterns, unsupported syntax.	The `BundlerOptions` includes flags like `strict: boolean`, `onwarn: (warning) => void`, and `maxCircularDepth: number`. The `ModuleGraph` and `Parser` consult these options to escalate warnings to errors or suppress them.	The user can customize the bundler's strictness via configuration. For example, setting `strict: true` turns all warnings into errors, failing the build. The `onwarn` callback allows custom logging or suppression.
Graceful Degradation	Source Map Generation Failure, partial tree-shaking side effect uncertainty.	The component catches the internal error, logs a warning, and provides a valid fallback. For source maps, if a mapping fails, we skip that mapping but still generate a partial source map. For side effects, we can conservatively keep code we're unsure about.	The build produces a functional artifact, but with a caveat (e.g., "Source map for `utils.js` is incomplete"). The user is informed of the compromise and can decide if action is needed.
Caching and Retry	Transient File System Errors (e.g., `EBUSY`).	The `FSCache.readFile` wrapper could implement a simple retry with exponential backoff for certain error codes before giving up and throwing.	For very rare edge cases, the build might succeed on a second automatic attempt, improving resilience in environments with aggressive file locking.

Architecture Decision: Error Severity Classification

> **Decision: Three-Tier Severity Model**
>
> - **Context**: We need to categorize issues to decide between failing, warning, or ignoring.
> - **Options Considered**:
>   1. **Binary (Error/Warning)**: Simple but doesn't capture "info" level diagnostics.
>   2. **Four-tier (Fatal/Error/Warning/Info)**: Overly complex for our scope.
>   3. **Three-tier (Error/Warning/Info)**: Covers all needs: errors stop the build, warnings indicate potential problems, info gives feedback (e.g., "Tree shaking removed 50% of lodash").
> - **Decision**: Use `'error'`, `'warning'`, `'info'` severity levels in `StructuredError`.
> - **Rationale**: Three tiers are sufficient for an educational bundler. `'info'` severity is useful for reporting optimization results without cluttering the warning output. The classification is clear for implementation.
> - **Consequences**: All diagnostic messages must be assigned an appropriate severity. The CLI formatter will color and prefix messages differently for each level (e.g., red for error, yellow for warning, blue for info).

Common Pitfalls in Error Handling

⚠️ Pitfall: Swallowing Errors in Asynchronous Code

- **Description**: Using `.catch()` without re-throwing or logging inside a promise chain in the bundler pipeline, causing a failure to silently disappear and the build to succeed incorrectly.
- **Why it's wrong**: An unresolved module error could be swallowed, leading to a bundle that fails at runtime with no build-time indication. This is incredibly difficult to debug.
- **Fix**: Always ensure errors are propagated to the top-level `Bundler.build()`. Use `async/await` with `try/catch` and re-throw `StructuredError`, or ensure promise chains have a final `.catch()` that converts to a rejected promise.

⚠️ Pitfall: Non-Descriptive Error Messages

- **Description**: Throwing an error like `"Module not found"` without specifying the importing file, the specifier, and the resolution paths that were tried.
- **Why it's wrong**: The developer must manually trace the import chain to find the problem, wasting time.
- **Fix**: Always enrich errors with context. The `StructuredError` should include `importer`, `specifier`, and `message` that lists the directories searched. For example: `"Cannot find module './utils' imported from /project/src/app.js. Searched in: /project/src, /project/node_modules"`.

⚠️ Pitfall: Incorrect Error Recovery During Tree Shaking

- **Description**: When the bundler encounters an error while analyzing a module for side effects (e.g., a parse error in a function that may have side effects), it might skip that module and incorrectly mark it as side-effect-free.
- **Why it's wrong**: This could lead to the removal of crucial code, breaking the application. It's safer to assume a module has side effects if analysis fails.
- **Fix**: Implement a **conservative default**. If side effect detection fails for any reason (parse error, unsupported syntax), treat the module as if it has side effects (`sideEffects: true`). Log a warning so the user knows analysis was incomplete.

⚠️ Pitfall: Not Handling Circular Dependencies in Topological Sort

- **Description**: Implementing `ModuleGraph.topologicalSort()` using Kahn's algorithm without a mechanism to handle cycles, causing an infinite loop or incorrect ordering.
- **Why it's wrong**: Many real-world libraries have circular dependencies. The bundler must produce a bundle that executes correctly, or at least fails with a clear error if the cycle is problematic.
- **Fix**: Detect cycles and handle them. One approach is to collapse **strongly connected components** (SCCs) into a single node during sorting. The runtime module system must then support initializing modules within an SCC in a way that avoids `ReferenceError`. Provide a warning when cycles are detected, as they can be a code smell.

Implementation Guidance

This section provides concrete code foundations for implementing the error handling system described above. We'll focus on the `StructuredError` class, a diagnostic collector, and integration points in the pipeline.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Error Structure	Custom `StructuredError` class with basic fields	Implement the `Diagnostic` interface from TypeScript compiler API for potential IDE integration
Error Formatting	Custom `formatDiagnostic()` function with colors via `chalk`	Use a library like `signale` or `listr2` for interactive, formatted output
Error Recovery	Simple try-catch with severity-based decision logic	Implement a `DiagnosticBag` with error budget and configurable thresholds

B. Recommended File/Module Structure

Add error handling types and utilities to a dedicated module.

```
build-your-own-bundler/
  src/
    bundler/
      index.ts                # Main Bundler class
      options.ts              # BundlerOptions type
    errors/                   # Error handling module
      index.ts                # Exports StructuredError, DiagnosticBag
      structured-error.ts     # StructuredError class definition
      diagnostic-bag.ts       # DiagnosticBag collector class
      codes.ts                # ERROR_CODES constant definitions
      formatter.ts            # CLI formatting utilities
    # ... other components (parser/, resolver/, graph/, generator/)
```

C. Infrastructure Starter Code

Here is complete, ready-to-use code for the core error handling infrastructure.

`src/errors/structured-error.ts`

```typescript
import type { BundlerOptions } from '../bundler/options';

/**
 * Defines the severity level of a diagnostic message.
 */
export type DiagnosticSeverity = 'error' | 'warning' | 'info';

/**
 * A structured diagnostic message used throughout the bundler pipeline.
 * Follows the naming convention `StructuredError`.
 */
export class StructuredError extends Error {
  public readonly code: string;
  public readonly severity: DiagnosticSeverity;
  public readonly location: { file: string; line: number; column: number } | null;
  public readonly specifier: string | null;
  public readonly importer: string | null;
  public readonly suggestions: string[];
  public readonly cause: Error | null;

  constructor(params: {
    code: string;
    message: string;
    severity?: DiagnosticSeverity;
    location?: { file: string; line: number; column: number } | null;
    specifier?: string | null;
    importer?: string | null;
    suggestions?: string[];
    cause?: Error | null;
  }) {
    super(params.message);
    this.name = 'StructuredError';
    this.code = params.code;
    this.severity = params.severity || 'error';
    this.location = params.location || null;
    this.specifier = params.specifier || null;
    this.importer = params.importer || null;
    this.suggestions = params.suggestions || [];
```

```
    this.cause = params.cause || null;

    // Ensure proper prototype chain for instanceof checks
    Object.setPrototypeOf(this, StructuredError.prototype);
  }

  /**
   * Converts the error to a plain object for JSON serialization (e.g., for IDE tools).
   */
  toJSON() {
    return {
      code: this.code,
      message: this.message,
      severity: this.severity,
      location: this.location,
      specifier: this.specifier,
      importer: this.importer,
      suggestions: this.suggestions,
      stack: this.stack,
    };
  }
}
```

src/errors/diagnostic-bag.ts

```typescript
import type { StructuredError } from './structured-error';
import type { BundlerOptions } from '../bundler/options';

/**
 * Collects diagnostics during a build and applies filtering/severity rules.
 */
export class DiagnosticBag {
  private diagnostics: StructuredError[] = [];

  /**
   * Adds a diagnostic to the bag. If options.strict is true, warnings are upgraded to errors.
   */
  add(diagnostic: StructuredError, options: BundlerOptions): void {
    let finalSeverity = diagnostic.severity;

    // Apply strict mode: upgrade warnings to errors
    if (options.strict && finalSeverity === 'warning') {
      finalSeverity = 'error';
    }

    // Apply custom warning handler
    if (finalSeverity === 'warning' && options.onwarn) {
      options.onwarn(diagnostic);
      // If onwarn doesn't throw, we still collect it
    }

    // Create a new diagnostic with potentially updated severity
    const finalDiagnostic = new StructuredError({ ...diagnostic, severity: finalSeverity });
    this.diagnostics.push(finalDiagnostic);
  }

  /**
   * Returns true if there are any errors (severity === 'error') in the bag.
   */
  hasErrors(): boolean {
    return this.diagnostics.some(d => d.severity === 'error');
  }
```

```
  /**
   * Returns all diagnostics, grouped by severity.
   */
  getAll(): { errors: StructuredError[]; warnings: StructuredError[]; infos: StructuredError[] } {
    return {
      errors: this.diagnostics.filter(d => d.severity === 'error'),
      warnings: this.diagnostics.filter(d => d.severity === 'warning'),
      infos: this.diagnostics.filter(d => d.severity === 'info'),
    };
  }

  /**
   * Clears all collected diagnostics.
   */
  clear(): void {
    this.diagnostics = [];
  }
}
```

`src/errors/codes.ts`

```typescript
/**
 * Standard error codes for the bundler.
 * Follows the naming convention `ERROR_CODES`.
 */
export const ERROR_CODES = {
  // Resolution errors
  MODULE_NOT_FOUND: 'MODULE_NOT_FOUND',
  PACKAGE_JSON_NOT_FOUND: 'PACKAGE_JSON_NOT_FOUND',
  PACKAGE_JSON_INVALID: 'PACKAGE_JSON_INVALID',
  EXPORTS_FIELD_RESOLUTION_FAILED: 'EXPORTS_FIELD_RESOLUTION_FAILED',
  SYMLINK_LOOP_DETECTED: 'SYMLINK_LOOP_DETECTED',

  // Parse & Syntax errors
  PARSE_ERROR: 'PARSE_ERROR',
  UNSUPPORTED_SYNTAX: 'UNSUPPORTED_SYNTAX',

  // Graph & Analysis errors
  CIRCULAR_DEPENDENCY_WARNING: 'CIRCULAR_DEPENDENCY_WARNING',
  EXPORT_NOT_FOUND: 'EXPORT_NOT_FOUND',
  IMPORT_BINDING_MISMATCH: 'IMPORT_BINDING_MISMATCH',

  // File system errors
  FILE_READ_ERROR: 'FILE_READ_ERROR',
  FILE_WRITE_ERROR: 'FILE_WRITE_ERROR',

  // Code generation errors
  SOURCE_MAP_GENERATION_ERROR: 'SOURCE_MAP_GENERATION_ERROR',

  // Internal errors
  INTERNAL_ERROR: 'INTERNAL_ERROR',
} as const;
```

D. Core Logic Skeleton Code

Integration of error handling into the main bundler pipeline and the resolver.

`src/bundler/index.ts` - Main pipeline with error handling

```typescript
import { DiagnosticBag } from '../errors/diagnostic-bag';
import { StructuredError } from '../errors/structured-error';
import { ERROR_CODES } from '../errors/codes';
import type { BundlerOptions } from './options';
import type { BundleChunk } from '../graph/types';
import { GraphBuilder } from '../graph/builder';
import { generateBundle } from '../generator';

export class Bundler {
  private options: BundlerOptions;
  private diagnostics: DiagnosticBag;

  constructor(options: BundlerOptions) {
    this.options = options;
    this.diagnostics = new DiagnosticBag();
  }

  /**
   * Main bundling pipeline execution.
   * Follows the naming convention `Bundler.build()`.
   */
  async build(): Promise<BundleChunk[]> {
    this.diagnostics.clear();

    try {
      // 1. Build the module graph
      const graphBuilder = new GraphBuilder(this.options, this.diagnostics);
      const moduleGraph = await graphBuilder.build(this.options.entryPoints);

      // 2. Check for errors collected during graph building
      if (this.diagnostics.hasErrors()) {
        // TODO: Format and throw the first error, or aggregate?
        const { errors } = this.diagnostics.getAll();
        throw errors[0];
      }

      // 3. Generate the bundle from the graph
      const chunks = await generateBundle(moduleGraph, this.options, this.diagnostics);
```

```
    // 4. If there are errors from generation, throw
    if (this.diagnostics.hasErrors()) {
      const { errors } = this.diagnostics.getAll();
      throw errors[0];
    }

    // 5. Print warnings and info (if any)
    this.printDiagnostics();

    return chunks;
  } catch (error) {
    // If it's already a StructuredError, re-throw
    if (error instanceof StructuredError) {
      throw error;
    }
    // Otherwise, wrap unknown errors
    throw new StructuredError({
      code: ERROR_CODES.INTERNAL_ERROR,
      message: `Internal bundler error: ${error.message}`,
      severity: 'error',
      cause: error instanceof Error ? error : null,
    });
  }
}

private printDiagnostics(): void {
  const { warnings, infos } = this.diagnostics.getAll();

  // Print warnings
  if (warnings.length > 0) {
    console.warn(`\n⚠️  Found ${warnings.length} warning(s):`);
    warnings.forEach(w => console.warn(formatWarning(w)));
  }

  // Print info messages
  if (infos.length > 0 && this.options.verbose) {
```

```
      console.info(`\nℹ️  ${infos.length} info message(s):`);
      infos.forEach(i => console.info(formatInfo(i)));
    }
  }
}

// Helper formatting functions (simplified)
function formatWarning(error: StructuredError): string {
  const loc = error.location ? ` at ${error.location.file}:${error.location.line}:${error.location.column}` : '';
  return `  • ${error.code}${loc}: ${error.message}`;
}

function formatInfo(error: StructuredError): string {
  return `  • ${error.message}`;
}
```

`src/resolver/index.ts` - Error handling in module resolution

TYPESCRIPT

```typescript
import { StructuredError } from '../errors/structured-error';
import { ERROR_CODES } from '../errors/codes';
import type { ResolutionContext } from './types';

/**
 * Main resolution method.
 * Follows the naming convention `resolve(specifier, fromDir)`.
 */
export async function resolve(
  specifier: string,
  fromDir: string,
  context: ResolutionContext
): Promise<string> {
  try {
    // TODO 1: Check if specifier is relative (./) or absolute (/)
    // TODO 2: For relative/absolute, call _resolvePath()
    // TODO 3: For bare specifier, call _resolveBareSpecifier()
    // TODO 4: Return the resolved absolute file path


    // Example error throwing for unresolved module:
    // throw new StructuredError({
    //   code: ERROR_CODES.MODULE_NOT_FOUND,
    //   message: `Cannot find module '${specifier}'`,
    //   severity: 'error',
    //   specifier,
    //   importer: fromDir,
    //   suggestions: [
    //     `Check the file exists at the resolved path.`,
    //     `If it's a package, ensure it's installed in node_modules.`
    //   ]
    // });

    return ''; // placeholder
  } catch (error) {
    // Enhance low-level filesystem errors with more context
    if (error.code === 'ENOENT') {
      throw new StructuredError({
```

```
        code: ERROR_CODES.MODULE_NOT_FOUND,
        message: `File not found: ${error.path}`,
        severity: 'error',
        location: null,
        specifier,
        importer: fromDir,
        cause: error,
      });
    }
    // Re-throw if already a StructuredError
    if (error instanceof StructuredError) {
      throw error;
    }
    // Wrap unknown errors
    throw new StructuredError({
      code: ERROR_CODES.INTERNAL_ERROR,
      message: `Resolution failed for '${specifier}' from ${fromDir}: ${error.message}`,
      severity: 'error',
      specifier,
      importer: fromDir,
      cause: error instanceof Error ? error : null,
    });
  }
}
```

E. Milestone Checkpoint

To verify your error handling is working, create a test project with intentional errors and run your bundler.

1. **Create a test file with an unresolved import**:

```
// src/test-error.js
import { missing } from './non-existent-module';
console.log('This should not run');
```
JAVASCRIPT

2. **Run your bundler** (assuming you have a CLI):

```
node your-bundler.js --entry ./src/test-error.js --out-dir dist
```
BASH

3. **Expected Behavior**:
   - The bundler should exit with a non-zero exit code.
   - A formatted error should print to stderr, resembling:

```
❌ MODULE_NOT_FOUND: Cannot find module './non-existent-module'
    imported from /project/src/test-error.js:1:20

Suggestions:
  • Check the file exists at the resolved path.
  • If it's a package, ensure it's installed in node_modules.
```

- No bundle file should be created in `dist/`.

4. **Signs of Problems**:
   - If the build succeeds silently, errors are being swallowed somewhere in the pipeline.
   - If the error message lacks the importing file location, the `importer` field isn't being set in the `StructuredError`.
   - If the process exits with code 0, ensure your CLI calls `process.exit(1)` when `Bundler.build()` throws.

F. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Build succeeds but runtime error "Cannot find module"	Error swallowed during resolution or graph building.	Add console.log or debugger statements in `resolve()` to see if it throws. Check if `DiagnosticBag.hasErrors()` is being called before generating the bundle.	Ensure all async operations in the pipeline are awaited and errors propagated. Wrap resolver/parser calls in try-catch that creates `StructuredError`.
Error message missing file/line information	`location` object not being constructed when creating `StructuredError`.	Inspect where the error originates (e.g., parser). The AST node has location info; ensure it's extracted and passed to the error constructor.	In `parseModule()`, catch the parser's syntax error and convert its location format to your `{file, line, column}` structure.
Warnings not appearing in output	Warnings are being collected but not printed, or `severity` is incorrectly set.	Check the `DiagnosticBag.getAll()` output after a build with known warnings (e.g., unused import). Verify the `severity` field is `'warning'`.	Ensure `printDiagnostics()` is called after a successful build, and that warnings aren't being filtered out by an `onwarn` handler that suppresses them.
Process hangs indefinitely on error	An error is thrown in an async callback without rejecting the promise, or a promise is never settled.	Use Node.js debugger or add logging at each pipeline stage to see where execution stops. Look for missing `await` or uncaught promise rejections.	Wrap entire pipeline in a timeout for debugging. Ensure all promises have `.catch()` handlers or use `async/await` with proper try-catch.

Testing Strategy

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

A robust testing strategy is essential for building a complex system like a bundler, where numerous interacting components must work together precisely. The primary challenge is validating that the transformation pipeline—from source code parsing through resolution, graph analysis, and code generation—produces correct, runnable output that faithfully preserves the semantics of the original modular program. Testing is especially critical because errors can manifest subtly, such as incorrectly resolved imports or missing tree-shaken code that breaks at runtime. This section outlines a comprehensive approach combining unit tests for isolated components, integration tests for the complete pipeline, and specific checkpoints for each milestone to verify steady progress.

Testing Approach

Think of testing a bundler as conducting a **multi-layered quality inspection** on an assembly line. At the first station, you examine individual machine parts (unit tests for parsers and resolvers). Next, you test subsystems like the engine (integration tests for the module graph). Finally, you take a finished car for a road test (end-to-end bundle execution) to ensure everything works together. This layered approach isolates failures and builds confidence incrementally.

We recommend a hybrid testing strategy with three primary pillars:

1. **Unit Tests for Isolated Components**: Test individual functions and classes in isolation, mocking external dependencies like the file system. This is ideal for the `parseModule`, `extractDependencies`, `resolve`, and `ModuleGraph` algorithms.
2. **Integration Tests with Fixture Projects**: Create small, self-contained test projects in a temporary directory that mimic real-world scenarios (e.g., a project with node_modules, various import styles, and package.json configurations). Run the full bundler on these fixtures and validate the output.
3. **Snapshot Testing for Generated Bundles**: For the final bundled code, use snapshot testing to detect unintended changes. This captures the generated bundle (and source map) as a "golden" reference and alerts you when the output changes.

Additionally, we employ **property-based testing** for the resolver to ensure it matches Node.js's behavior across many random specifier and directory structures, and **visualization aids** like logging the module graph for manual inspection during debugging.

Core Testing Infrastructure

To support these testing styles, we define a few key utilities. The `FSCache` can be replaced with an in-memory mock for unit tests, allowing precise control over file system state. A `DiagnosticBag` collects errors and warnings, enabling tests to assert on specific error codes and messages. For integration tests, a helper creates a temporary directory tree, writes test files, runs the bundler, and cleans up.

Utility Component	Purpose in Testing	Implementation Notes
`MockFileSystem`	Provides an in-memory replacement for `fs` operations, allowing tests to define virtual directory structures without touching the disk.	Extends `FSCache` interface; stores files in a `Map<string, string>`; simulates `readFile`, `stat`, `readdir`.
`FixtureBuilder`	Programmatically creates temporary test projects on disk for integration tests.	Uses `os.tmpdir()`; creates folders, `package.json` files, and source modules; returns path to cleanup function.
`SnapshotManager`	Manages golden snapshots for bundle output and source maps.	Stores snapshots in `__snapshots__/` directory; uses a deterministic formatting for ASTs and bundles; provides diff output on mismatch.
`DiagnosticCollector`	Wraps `DiagnosticBag` for easier assertion in tests.	Methods like `expectError(code)`, `expectWarning(pattern)`, `assertNoErrors()`.

Test Organization

The test suite mirrors the project's architecture, with a test file for each major component and a separate directory for integration fixtures.

```
project-root/
├── src/
│   ├── parser/          # Component 1
│   ├── resolver/        # Component 2
│   ├── graph/           # Component 3
│   └── generator/       # Component 4
└── tests/
    ├── unit/
    │   ├── parser.test.js
    │   ├── resolver.test.js
    │   ├── graph.test.js
    │   └── generator.test.js
    ├── integration/
    │   ├── fixtures/    # Each subdir is a test project
    │   │   ├── simple-esm/
    │   │   ├── commonjs-interop/
    │   │   └── node-modules-resolution/
    │   └── bundler.test.js
    └── utils/           # MockFileSystem, FixtureBuilder, etc.
```

Milestone Checkpoints

Each milestone has a concrete deliverable and corresponding acceptance criteria. The checkpoints below provide specific commands to run and observable outcomes to verify that the milestone has been successfully implemented. These checkpoints are cumulative; later milestones assume functionality from earlier ones.

Milestone 1: Module Parsing

Goal: Verify that the parser correctly reads source files, builds an AST, and extracts all static and dynamic dependencies.

Checkpoint Test Command:

```bash
npm test -- tests/unit/parser.test.js
```

Expected Success Output:

- All tests pass, showing that `parseModule` returns a `ModuleNode` with correct `imports`, `exports`, and `dynamicImports`.
- For a given source file, the `dependencies` map contains the exact specifiers found in the code.
- The `exports` set accurately lists all exported identifiers (named, default, and re-exports).
- Dynamic imports (`import()`) are captured in `dynamicImports` with their specifier and location.

Manual Verification Steps:

1. Create a test file `test-module.js`:

```javascript
import { foo } from './lib.js';

import * as bar from 'bar';

export const baz = foo + bar.qux;

export default function() {};

export { foo as reexportedFoo } from './lib.js';

import('./lazy.js').then(m => m.load());
```

2. Run your parser on this file and inspect the resulting `ModuleNode`.
3. Validate:
   - `dependencies` has entries for `'./lib.js'` and `'bar'`.
   - `exports` includes `'baz'`, `'default'`, and `'reexportedFoo'`.
   - `dynamicImports` contains one entry with specifier `'./lazy.js'`.
   - The `imports` map details each import's `local` name and `imported` name.

Common Failure Signs:

- Missing dependencies: Often caused by not handling all AST node types (e.g., `ExportNamedDeclaration` with `source`).
- Incorrect specifier: Extracting the wrong string from the AST node.
- No dynamic imports captured: Forgetting to check `ImportExpression` nodes (Babel's AST type for `import()`).

Milestone 2: Module Resolution

Goal: Verify that the resolver correctly translates module specifiers to absolute file paths following Node.js rules.

Checkpoint Test Command:

```bash
npm test -- tests/unit/resolver.test.js
```

Expected Success Output:

- All unit tests pass, covering relative paths (`./file`), absolute paths, and bare specifiers (`lodash`).
- Tests for `package.json` field priority (`exports` > `module` > `main`) pass.
- The resolver correctly traverses parent directories to find `node_modules`.

Integration Test:

1. Create a fixture project with a `node_modules` containing a package `my-pkg` with `package.json` specifying `"main": "lib/index.js"` and an `"exports"` field.
2. Run the bundler's resolution on an entry file that imports `'my-pkg'`.

3. Verify the resolved path points to the correct file as per Node.js rules.

Key Behavior to Validate:

- `resolve('./lib', '/project/src')` → `/project/src/lib.js` (with automatic extension).
- `resolve('lodash', '/project/src')` → traverses up to find `/project/node_modules/lodash` and resolves via its `package.json`.
- Conditional exports in `package.json` are respected (e.g., `require` vs `import` conditions).

Common Failure Signs:

- Infinite loops: Occur when symlinks are not tracked in `ResolutionContext.visitedSymlinks`.
- Wrong file resolved: Incorrect order of `package.json` field lookup or mis-handling of directory `index.js` fallback.
- Slow resolution: Not using `FSCache` or `resolutionCache` leads to repeated file system reads.

Milestone 3: Bundle Generation

Goal: Verify that the bundler produces a single, executable JavaScript file that correctly wires together all modules.

Checkpoint Test Command:

```bash
npm test -- tests/unit/generator.test.js && npm test -- tests/integration/bundler.test.js -t "simple bundle"
```

Expected Success Output:

- Unit tests for `rewriteModuleAST` demonstrate that import/export statements are transformed to calls to the runtime module registry.
- Integration test produces a bundle file that can be executed in Node.js (or a browser) without errors.
- The bundle includes the runtime loader code and wrapped module functions.

Manual Verification Steps:

1. Create a simple project with two modules: `index.js` imports from `math.js` and exports a result.
2. Run the bundler to produce `bundle.js`.
3. Execute `node bundle.js` and verify the expected output is printed.
4. Inspect the bundle code to confirm:
   - Each module is wrapped in a function (`function(module, exports, require) { ... }` or similar).
   - Import statements have been replaced with `require` calls.
   - Export statements assign to `module.exports`.
   - Modules are topologically sorted (dependencies before dependents).

Source Map Validation:

- If source maps are enabled, verify that a source map is emitted and that its `sources` array contains the original file paths.
- Use a source map visualizer or browser DevTools to confirm mappings are correct.

Common Failure Signs:

- Runtime error: `require is not defined` — the runtime loader is missing or incorrect.
- Wrong execution order: Modules trying to import from a dependency that hasn't been loaded yet (topological sort is wrong).
- Exports are `undefined`: Incorrect rewriting of export statements, especially default exports.

Milestone 4: Tree Shaking

Goal: Verify that unused exports are eliminated from the final bundle, and code with side effects is preserved.

Checkpoint Test Command:

```bash
npm test -- tests/unit/graph.test.js -t "shake" && npm test -- tests/integration/bundler.test.js -t "tree shaking"
```

Expected Success Output:

- Unit tests confirm that `ModuleGraph.markUsedExports` correctly marks exports reachable from the entry point and that `ModuleGraph.shake` removes unmarked modules and exports.

- Integration tests show that bundles for projects with unused exports are smaller and that side-effectful code (like `console.log` at the top level) is retained even if its module's exports are unused.

Manual Verification Steps:

1. Create a project with a module that exports multiple functions but only one is imported.
2. Build with tree shaking enabled.
3. Inspect the bundle: the unused export and its associated code should be absent.
4. Add a module with `sideEffects: false` in its `package.json` and no used exports. Verify the entire module is omitted from the bundle.
5. Add a module with a top-level `console.log('side effect')` but no used exports. Verify the log statement remains in the bundle.

Quantitative Check:

- The bundler should report statistics on removed modules and export counts (part of `BundleChunk` metadata).

Common Failure Signs:

- Necessary code removed: Side effect detection fails, removing code that modifies global state.
- Unused code not removed: The export marking algorithm didn't traverse re-export chains correctly.
- Incorrect handling of `sideEffects: false`: The field is ignored or applied too aggressively.

> **Key Insight:** The most reliable way to test tree shaking is to compare the behavior of the original unbundled code (run via Node with native modules) with the bundled output. They should produce identical side effects and return values.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Test Runner	Node.js built-in `assert` + simple test runner script	Jest or Vitest for rich features (snapshots, mocks, coverage)
Mock File System	Custom `MockFileSystem` class using `Map`	`memfs` library for complete `fs` API simulation
Snapshot Testing	Manual string comparison with golden files	Jest snapshot or `snap-shot-core` library
Property-Based Testing	Manual generation of random test cases	`fast-check` or `jsverify` for automated property testing

B. Recommended File/Module Structure

```
build-your-own-bundler/
├── src/
│   └── (as previously defined)
├── tests/
│   ├── unit/
│   │   ├── parser.test.js
│   │   ├── resolver.test.js
│   │   ├── graph.test.js
│   │   └── generator.test.js
│   ├── integration/
│   │   ├── fixtures/
│   │   │   ├── simple-esm/
│   │   │   │   ├── index.js
│   │   │   │   └── math.js
│   │   │   ├── commonjs-interop/
│   │   │   └── node-modules-resolution/
│   │   └── bundler.test.js
│   └── utils/
│       ├── mock-fs.js
│       ├── fixture-builder.js
│       └── diagnostic-collector.js
├── package.json
└── (configuration files)
```

C. Infrastructure Starter Code

Complete `MockFileSystem` implementation (mock-fs.js):

JAVASCRIPT

```javascript
/**
 * In-memory mock file system for testing.
 * Implements a subset of fs operations used by the resolver and parser.
 */
class MockFileSystem {
  constructor() {
    this.files = new Map(); // path -> content
    this.dirs = new Set(); // path strings for directories
  }

  addFile(path, content) {
    this.files.set(path, content);
    // Ensure parent directories exist
    const dir = this._dirname(path);
    this.dirs.add(dir);
    let parent = this._dirname(dir);
    while (parent && parent !== dir) {
      this.dirs.add(parent);
      dir = parent;
      parent = this._dirname(dir);
    }
  }

  addDirectory(path) {
    this.dirs.add(path.endsWith('/') ? path : path + '/');
  }

  async readFile(path, encoding = 'utf8') {
    if (!this.files.has(path)) {
      const err = new Error(`ENOENT: no such file or directory, open '${path}'`);
      err.code = 'ENOENT';
      throw err;
    }
    const content = this.files.get(path);
    return encoding === 'buffer' ? Buffer.from(content) : content;
  }

  async stat(path) {
```

```
  if (this.files.has(path)) {
    return { isFile: () => true, isDirectory: () => false };
  }
  if (this.dirs.has(path) || this.dirs.has(path + '/')) {
    return { isFile: () => false, isDirectory: () => true };
  }
  const err = new Error(`ENOENT: no such file or directory, stat '${path}'`);
  err.code = 'ENOENT';
  throw err;
}

async readdir(path) {
  const prefix = path.endsWith('/') ? path : path + '/';
  const fileNames = new Set();
  for (const filePath of this.files.keys()) {
    if (filePath.startsWith(prefix)) {
      const remaining = filePath.slice(prefix.length);
      const nextSlash = remaining.indexOf('/');
      if (nextSlash === -1) {
        fileNames.add(remaining);
      } else {
        fileNames.add(remaining.slice(0, nextSlash));
      }
    }
  }
  for (const dirPath of this.dirs) {
    if (dirPath.startsWith(prefix) && dirPath !== prefix) {
      const remaining = dirPath.slice(prefix.length);
      const nextSlash = remaining.indexOf('/');
      if (nextSlash !== -1) {
        fileNames.add(remaining.slice(0, nextSlash));
      }
    }
  }
  return Array.from(fileNames);
}
```

```
  _dirname(path) {
    const idx = path.lastIndexOf('/');
    return idx === -1 ? '' : path.slice(0, idx);
  }
}

module.exports = { MockFileSystem };
```

Diagnostic Collector utility (diagnostic-collector.js):

JAVASCRIPT

```javascript
const { DiagnosticBag, StructuredError } = require('../../src/shared/diagnostics');

class DiagnosticCollector {
  constructor() {
    this.bag = new DiagnosticBag();
  }

  add(diagnostic) {
    this.bag.add(diagnostic);
  }

  expectError(code, pattern) {
    const errors = this.bag.getAll().errors || [];
    const matching = errors.filter(e => e.code === code && (!pattern || e.message.match(pattern)));
    if (matching.length === 0) {
      const errorList = errors.map(e => `- ${e.code}: ${e.message}`).join('\n');
      throw new Error(`Expected error with code "${code}" but found:\n${errorList}`);
    }
  }

  expectWarning(pattern) {
    const warnings = this.bag.getAll().warnings || [];
    const matching = warnings.filter(w => w.message.match(pattern));
    if (matching.length === 0) {
      const warningList = warnings.map(w => `- ${w.message}`).join('\n');
      throw new Error(`Expected warning matching "${pattern}" but found:\n${warningList}`);
    }
  }

  assertNoErrors() {
    if (this.bag.hasErrors()) {
      const errors = this.bag.getAll().errors || [];
      const errorList = errors.map(e => `- ${e.code}: ${e.message}`).join('\n');
      throw new Error(`Unexpected errors:\n${errorList}`);
    }
  }

  clear() {
    this.bag.clear();
```

```
  }
}

module.exports = { DiagnosticCollector };
```

D. Core Logic Skeleton Code for Unit Tests

Example unit test for the resolver (`tests/unit/resolver.test.js`):

JAVASCRIPT

```javascript
const { describe, it, beforeEach } = require('@jest/core'); // or your test framework

const { MockFileSystem } = require('../utils/mock-fs');

const { createResolver } = require('../../src/resolver');


describe('Module Resolver', () => {

  let mockFs;

  let resolve;


  beforeEach(() => {

    mockFs = new MockFileSystem();

    // Create a resolver that uses our mock file system

    resolve = createResolver({

      extensions: ['.js', '.json'],

      mainFields: ['module', 'main'],

      fsCache: { readFile: (p) => mockFs.readFile(p) },

      // ... other resolver options

    });

  });


  it('resolves relative path with .js extension', async () => {

    mockFs.addFile('/project/src/index.js', '');

    mockFs.addFile('/project/src/utils.js', '');


    const resolved = await resolve('./utils', '/project/src');

    // TODO: Assert that resolved equals '/project/src/utils.js'

  });


  it('resolves bare specifier from node_modules', async () => {

    mockFs.addFile('/project/node_modules/lodash/package.json', JSON.stringify({

      name: 'lodash',

      main: 'lodash.js'

    }));

    mockFs.addFile('/project/node_modules/lodash/lodash.js', '');


    const resolved = await resolve('lodash', '/project/src');

    // TODO: Assert that resolved equals '/project/node_modules/lodash/lodash.js'

  });


  it('respects package.json exports field', async () => {

    mockFs.addFile('/project/node_modules/pkg/package.json', JSON.stringify({
```

```
      name: 'pkg',
      exports: {
        '.': './dist/index.js',
        './feature': './dist/feature.js'
      }
    }));
    mockFs.addFile('/project/node_modules/pkg/dist/index.js', '');
    mockFs.addFile('/project/node_modules/pkg/dist/feature.js', '');

    const resolved = await resolve('pkg/feature', '/project/src');
    // TODO: Assert that resolved equals '/project/node_modules/pkg/dist/feature.js'
  });

  // TODO: Add more test cases for error conditions, symlinks, index.js fallback, etc.
});
```

Example integration test using fixtures (`tests/integration/bundler.test.js`):

JAVASCRIPT

```javascript
const { Bundler } = require('../../src/bundler');
const { FixtureBuilder } = require('../utils/fixture-builder');
const path = require('path');

describe('Bundler Integration', () => {
  let fixtureBuilder;

  beforeEach(() => {
    fixtureBuilder = new FixtureBuilder();
  });

  afterEach(async () => {
    await fixtureBuilder.cleanup();
  });

  it('bundles a simple ESM project', async () => {
    const projectPath = await fixtureBuilder
      .addFile('index.js', `
        import { add } from './math.js';
        console.log(add(2, 3));
      `)
      .addFile('math.js', `
        export function add(a, b) { return a + b; }
        export function unused() {}
      `)
      .build();

    const bundler = new Bundler({
      entryPoints: [path.join(projectPath, 'index.js')],
      outDir: path.join(projectPath, 'dist'),
      sourceMap: false,
    });

    const chunks = await bundler.build();
    // TODO: Assert that one chunk was produced
    // TODO: Read the generated bundle file and assert it contains the expected code
    // TODO: Optionally execute the bundle in a child process and verify output
  });

  // TODO: Add more integration tests for CommonJS, tree shaking, code splitting, etc.
```

```
});
```

E. Language-Specific Hints (JavaScript/Node.js)

- **Mocking file system:** Use the `MockFileSystem` class above, or consider the `memfs` npm package for a more complete `fs` mock.
- **Snapshot testing:** Jest provides `toMatchSnapshot()`. For a manual approach, write the bundle output to a file in `__snapshots__` and use `fs.readFileSync` to compare.
- **Running generated bundles:** Use `child_process.execFile` with `node` to execute the bundle and capture stdout/stderr.
- **Temporary directories:** Use `os.tmpdir()` and `fs.mkdtempSync` to create isolated test directories.
- **AST comparison:** Use `JSON.stringify` with a deterministic replacer to compare ASTs in tests, or use a library like `ast-pretty-print`.

F. Milestone Checkpoint Commands

As you implement each milestone, run these commands to verify progress:

1. **Milestone 1:** `npm test -- tests/unit/parser.test.js` — All parser tests pass.
2. **Milestone 2:** `npm test -- tests/unit/resolver.test.js` — All resolver tests pass, and `npm test -- tests/integration/bundler.test.js -t "resolution"` passes.
3. **Milestone 3:** `npm test -- tests/unit/generator.test.js` and `npm test -- tests/integration/bundler.test.js -t "simple bundle"` — Bundle generation and execution tests pass.
4. **Milestone 4:** `npm test -- tests/unit/graph.test.js -t "shake"` and `npm test -- tests/integration/bundler.test.js -t "tree shaking"` — Tree shaking tests pass, and bundle size decreases appropriately for unused code.

G. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Parser test fails on a specific import syntax	AST visitor missing a node type (e.g., `ImportExpression` for dynamic import).	Log the AST node type of the problematic statement. Check if your visitor handles that type.	Add handler for the missing node type in `extractDependencies`.
Resolver enters infinite loop	Symlink cycle not detected.	Log `visitedSymlinks` in `ResolutionContext`. See if the same real path appears twice.	Ensure `visitedSymlinks` tracks canonical paths and checks before following symlinks.
Generated bundle throws `require is not defined`	Runtime loader code not included or `require` function not defined in wrapper.	Inspect the beginning of the bundle file. Is there a `(function(modules){...})` wrapper?	Ensure `generateBundle` includes the runtime loader and each module function receives `require` as a parameter.
Tree shaking removes necessary side effect	Module's `sideEffects` field incorrectly set to `false` or side effect detection fails.	Check the `package.json` of the affected module. Does it have `"sideEffects": false`?	For modules with side effects, set `"sideEffects": true` or remove the field (conservative default). Improve static analysis to detect top-level function calls.
Source map mappings are off by one line	Line/column accounting error during code generation.	Use a source map visualizer (e.g., `source-map-visualization`) to see mappings.	Ensure `SourceMapManager.addMapping` uses correct original and generated positions, accounting for added wrapper lines.

Debugging Guide

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

Debugging a bundler is particularly challenging because errors can originate at multiple layers: from incorrect static analysis during the parsing phase, faulty module resolution that misses dependencies, incorrect graph transformations during tree shaking, or runtime generation issues in the final bundle. This guide provides a structured approach to diagnosing and fixing the most common issues learners encounter when building their own JavaScript bundler. Think of debugging a bundler like being a **software detective**—you must systematically gather clues (logs, error messages,

generated code), reconstruct the crime scene (the module graph and transformations), and identify which component committed the "crime" (introduced the bug).

The key insight is that bundlers follow a **unidirectional pipeline pattern**—errors cascade downstream. A parsing error in Component 1 will manifest as missing dependencies in Component 2, which causes incomplete graph building in Component 3, resulting in runtime errors in Component 4. Therefore, the first step in debugging is always to isolate which component is responsible by checking the data at each stage boundary.

Common Bug Table

The following table catalogs the most frequent symptoms learners encounter, organized by the phase of the bundling pipeline where they typically originate. Each entry includes the **symptom** (what you observe), the **likely cause** (which component and specific logic is faulty), and the **fix** (concrete steps to correct the issue).

Symptom	Likely Cause	Fix
Runtime Error: `Cannot read property 'default' of undefined`	The module runtime registry doesn't have an entry for the requested module ID because the module wasn't included in the bundle. This typically occurs when tree shaking incorrectly prunes a module that's actually used, or when module resolution failed to locate a dependency.	1. Check the `ModuleGraph.shake()` logic to ensure you're marking all used exports from entry points correctly. 2. Verify module resolution succeeded for all imports by logging the `dependencies` map in each `ModuleNode`. 3. Ensure circular dependencies are handled correctly—modules involved in cycles must all be included.
Runtime Error: `Module not found: Error: Can't resolve './utils'`	The module specifier wasn't resolved to an absolute path during the resolution phase. This happens when `resolve()` returns `null` or throws for a relative import, often due to missing file extensions or incorrect `package.json` field lookup.	1. Add logging in `resolve()` to trace the resolution steps for the problematic specifier. 2. Ensure your resolver checks all valid extensions (`.js` , `.ts` , `.json`) and the `index.js` fallback. 3. Verify the `fromDir` parameter is correctly set to the importing module's directory.
Bundle executes in wrong order, causing undefined variable errors	The topological sort of the module graph is incorrect, placing a dependent module before its dependency. This often happens when the dependency graph has cycles that aren't handled properly, or when the sort algorithm doesn't use a **post-order dependency order**.	1. Implement cycle detection using Tarjan's algorithm for **strongly connected components** and treat each SCC as a single unit in topological order. 2. Verify that `ModuleGraph.linkDependency()` correctly establishes bidirectional edges. 3. Use Kahn's algorithm with a worklist and track indegrees, handling cycles by breaking them arbitrarily.
Tree shaking removes code that has side effects (e.g., polyfills)	The bundler incorrectly assumes a module is side-effect-free. This occurs when you don't respect the `sideEffects: false` field in `package.json` , or when your static analysis fails to detect implicit side effects like top-level function calls or assignments to global variables.	1. Always read the `sideEffects` field from `package.json` and set `ModuleNode.sideEffects` accordingly (default to `true` for **conservative default**). 2. Enhance side effect detection in the parser to flag any statement that isn't an import/export declaration or a function/class declaration. 3. When in doubt, preserve the entire module.
Source maps don't line up—debugger jumps to wrong lines	Mappings in the generated source map are incorrect. This usually happens because positions aren't tracked correctly during AST transformation, or because the **source map mappings** VLQ encoding is off by one.	1. Use a `SourceMapManager` to track positions consistently. 2. After each AST transformation, update mapping positions to account for added/removed characters. 3. Verify that `sourcesContent` matches the original source exactly, including whitespace.
Exported live bindings don't update—imports stay at initial value	The runtime wrapper doesn't implement **live bindings** correctly. In ESM, exports are live references, not copies. Your generated code might be copying the initial value rather creating a getter that reads from the original variable.	1. In the module wrapper, export an object with getter functions or use `Object.defineProperty` for each export. 2. Ensure re-exports (`export * from './foo'`) create indirect references rather than copying values. 3. Test with a cyclic dependency where one module mutates an export after initialization.
Dynamic imports (`import()`) cause runtime errors or aren't code-split	The parser didn't detect dynamic imports, or the code generator didn't create separate **bundle chunks**. Dynamic imports require special handling: they're code-split points that generate separate files loaded at runtime.	1. In `extractDependencies()` , look for `ImportExpression` nodes (Babel) or `CallExpression` with `callee.type === 'Import'` . 2. Mark these in `ModuleNode.dynamicImports` . 3. In code generation, create a new `BundleChunk` for each unique dynamic import target reachable from entry points.
Bare specifiers like `lodash` resolve to wrong file	The `_resolveBareSpecifier()` algorithm doesn't correctly traverse parent directories or doesn't respect the `exports` field in `package.json` . Node's resolution is complex, with `exports` taking precedence over `main` .	1. Ensure your resolver walks up directories until root, checking `node_modules/package` at each level. 2. Implement `_resolveExports()` to handle the `exports` map, including conditional exports like `import` / `require` . 3. Test with a nested `node_modules` structure.
Circular dependencies cause infinite loops in graph traversal	The graph building uses naive recursion without cycle detection, causing infinite recursion or stack overflow. This is common when traversing dependencies using DFS without a `visited` set.	1. Use a `Set` to track visited module IDs during graph construction. 2. When building the graph via BFS, use a worklist algorithm and avoid reprocessing already-visited nodes. 3. Consider using an iterative approach instead of recursion for dependency collection.

Symptom	Likely Cause	Fix
Re-export chains break—exported symbol is `undefined`	The bundler doesn't properly follow **re-export chains**. When module A does `export { x } from './b'` and module B does `export { y as x } from './c'`, you need to trace through multiple hops to find the original source.	1. During graph building, expand re-exports by following the chain to the original export. 2. Store re-export mappings in `ModuleNode.reexports` with `{source, exported}`. 3. During tree shaking, mark the original export as used when a re-export is used.
TypeScript files aren't parsed (`SyntaxError: Unexpected token`)	The parser isn't configured to handle TypeScript syntax. You're likely using a JavaScript-only parser without the TypeScript plugin enabled.	1. Configure `@babel/parser` with the `plugins: ['typescript']` option. 2. Alternatively, use the TypeScript compiler API (`ts.createSourceFile`) for `.ts` files. 3. Ensure file extension detection includes `.ts` and `.tsx`.
Bundle size is much larger than expected—no tree shaking	The `markUsedExports()` algorithm isn't traversing the entire graph, or it's only marking exports explicitly imported by name, missing namespace (`*`) or re-export usage.	1. Ensure marking starts from all entry points and follows all import types (named, default, namespace). 2. When a namespace import (`import * as foo`) is used, mark **all** exports of the target module as used. 3. Verify that `sideEffects: false` modules are being removed entirely when no exports are used.
`require()` calls (CommonJS) aren't recognized as dependencies	The parser only looks for ES module `import` statements and misses `CallExpression` nodes with `callee.name === 'require'`.	1. In `extractDependencies()`, add a case for `CallExpression` with `callee.type === 'Identifier'` and `callee.name === 'require'`. 2. Extract the string argument as a dependency with type `'static'`. 3. Note: Dynamic `require` (variable argument) is harder—consider assuming it's side-effectful.
Generated bundle pollutes global scope with variables	Module wrapping is missing or incorrect. Each module's top-level code should be inside a function scope to isolate its variables.	1. Ensure `generateBundle()` wraps each module's transformed AST in a function expression that's immediately invoked or registered with the runtime. 2. Use strict mode (`'use strict'`) inside the wrapper to catch accidental globals. 3. Check that `var` declarations inside the module don't become global.
Symlinks cause infinite resolution loops	The resolver doesn't detect symlink cycles, leading to infinite recursion when following symlinks that point to ancestors in the directory tree.	1. In `ResolutionContext`, maintain a `visitedSymlinks` set of realpath strings. 2. Before following a symlink, check if its realpath is already in the set—if yes, throw a **structured error** about circular symlinks. 3. Use `fs.realpath` to resolve symlinks to their canonical form.

Debugging Techniques

When the above table doesn't match your specific symptom, you'll need to employ systematic debugging techniques to isolate the issue. The following approaches are ranked from quickest to most thorough.

1. Inspect the Module Graph

The **module graph** is the single most important data structure for debugging. When anything goes wrong, dump the graph to console to see if dependencies are missing, cycles exist, or tree shaking marks are incorrect.

How to implement logging:

```javascript
// In your ModuleGraph class, add a debug method

debugPrint() {

  for (const [id, node] of this.nodes) {

    console.log(`Module: ${id}`);

    console.log(`  Dependencies: ${[...node.dependencies.keys()]}`);

    console.log(`  Dependents: ${[...node.dependents]}`);

    console.log(`  Exports: ${[...node.exports]}`);

    console.log(`  Used Exports: ${[...node.usedExports]}`);

    console.log(`  Side effects: ${node.sideEffects}`);

    console.log(`  Is entry: ${node.isEntry}`);

  }

}
```

Call this after each major phase: after graph building, after marking used exports, and after shaking. Look for:

- **Missing dependencies**: A module lists an import, but the dependency isn't in the graph.
- **Unmarked used exports**: An export that should be reachable isn't in `usedExports`.
- **Unexpected dependents**: A module that shouldn't be included has dependents.

2. Trace Module Resolution

When a module can't be found, enable verbose logging in the `resolve()` function to see each step of the resolution algorithm.

What to log:

- The specifier and `fromDir` at the start.
- Each directory checked for `node_modules`.
- Each `package.json` read and the fields considered.
- The final resolved path or `null`.

Example output:

```
Resolving 'lodash' from '/project/src'
  Checking /project/src/node_modules/lodash → not found
  Checking /project/node_modules/lodash → found package dir
  Read package.json: main='lodash.js', exports={...}
  Resolved via exports to '/project/node_modules/lodash/lodash.js'
```

This reveals whether the resolver is looking in the right places and interpreting `package.json` correctly.

3. Examine the Generated AST

When the bundle behaves incorrectly (wrong values, missing code), inspect the transformed AST of a specific module to see if import/export rewriting is correct.

Technique: Use a code generator like `@babel/generator` to convert the transformed AST back to source, then compare with the original.

```javascript
const generate = require('@babel/generator').default;

const transformedCode = generate(transformedAST).code;

console.log(transformedCode);
```

Look for:

- **Import statements** that weren't replaced with runtime calls (e.g., `require__`).

- **Export statements** that weren't converted to assignments to `module.exports`.
- **Incorrect variable renaming** that breaks scoping.

4. Use the Node.js Debugger

For runtime errors in the generated bundle, use the Node.js inspector to set breakpoints in the **generated bundle code**. This is especially useful for debugging the runtime module registry.

Steps:

1. Run Node with `node --inspect-brk dist/bundle.js`.
2. Open Chrome DevTools at `chrome://inspect`.
3. Navigate to the generated bundle (it'll be a single large file).
4. Set breakpoints in the runtime loader function (look for `function require(moduleId)`).
5. Step through to see which module fails to load or returns undefined exports.

Pro tip: Add a source map to make debugging easier—you'll see original source lines instead of generated code.

5. Create Minimal Test Cases

When you encounter a complex bug, isolate it by creating the smallest possible project that reproduces the issue.

Example structure:

```
test-case/
  package.json
  src/
    index.js
    utils.js
```

Write minimal modules that demonstrate the bug (e.g., a circular dependency, a re-export chain, a dynamic import). Then run your bundler on this test case and observe where it fails. This eliminates noise from your main project.

6. Validate Each Pipeline Stage

Since the bundler is a pipeline, you can validate the output of each component independently.

Stage	Validation Method
Parsing	Write a test that parses a known file and asserts the extracted dependencies match expectations. Use snapshot testing for the AST.
Resolution	Test `resolve()` with various specifiers and expected outcomes. Use a **mock file system** to simulate `node_modules` structures.
Graph Building	After building the graph from a small fixture, assert the number of nodes and edges. Check that all dependencies are bidirectional.
Tree Shaking	Create a module with used and unused exports, run marking, and verify that `usedExports` is correct.
Code Generation	Generate a bundle and run it in Node, asserting the exported values are correct. Use **property-based testing** to generate random module graphs and verify the bundle executes without errors.

7. Check for Silent Failures

Some errors don't throw but produce incorrect bundles. Always enable strict error checking:

- Ensure `DiagnosticBag.hasErrors()` is checked after each phase.
- Convert warnings to errors during development to catch issues early.
- Use `try/catch` around file system operations and AST transformations, converting to **structured errors** with context (file path, specifier).

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Debugging Logging	`console.log` with conditional flags	Structured logging with `pino` / `winston`, log levels, and JSON output
Module Graph Visualization	Text dump to console	Generate DOT file for Graphviz, or use `vis-network` for browser visualization
Source Map Inspection	Manual inspection of mappings	Use `source-map` library to parse and query mappings programmatically
AST Inspection	`console.log(JSON.stringify(ast, null, 2))`	Use `@babel/traverse` to write custom debug visitors, or AST explorer web tools

B. Recommended File/Module Structure

Add debugging utilities in a dedicated directory:

```
bundler/
  src/
    debug/                    ← Debugging utilities
      index.js                ← Main debug exports
      graph-logger.js         ← Module graph logging
      resolution-tracer.js    ← Resolution step logging
      ast-inspector.js        ← AST pretty-printing
    diagnostics/              ← Structured errors (from earlier)
      diagnostic-bag.js
      structured-error.js
    ... (other components)
```

C. Infrastructure Starter Code

Here's a complete, ready-to-use **module graph debug logger** that prints the graph in a readable format:

JAVASCRIPT

```javascript
// src/debug/graph-logger.js

const chalk = require('chalk'); // Optional: for colored output


class GraphLogger {

  /**

   * Prints the entire module graph to console, highlighting entry points,

   * used/unused exports, and side-effect flags.

   */

  static printGraph(moduleGraph, options = {}) {

    const { showDependents = true, showUnused = true } = options;



    console.log(chalk.bold.blue('\n=== MODULE GRAPH ===\n'));



    for (const [id, node] of moduleGraph.nodes) {

      const entryTag = node.isEntry ? chalk.green(' [entry]') : '';

      const sideEffectTag = node.sideEffects ? chalk.yellow(' [side-effects]') : '';

      console.log(chalk.bold(`${id}${entryTag}${sideEffectTag}`));



      // Dependencies

      if (node.dependencies.size > 0) {

        console.log('  Dependencies:');

        for (const [specifier, depId] of node.dependencies) {

          console.log(`    ${specifier} → ${depId}`);

        }

      }



      // Dependents

      if (showDependents && node.dependents.size > 0) {

        console.log('  Dependents:');

        for (const depId of node.dependents) {

          console.log(`    ← ${depId}`);

        }

      }



      // Exports

      if (node.exports.size > 0) {

        console.log('  Exports:');
```

```
        for (const exp of node.exports) {
          const isUsed = node.usedExports.has(exp);
          const usedMarker = isUsed ? chalk.green(' ✓') : (showUnused ? chalk.gray(' ✗') : '');
          console.log(`    ${exp}${usedMarker}`);
        }
      }

      console.log(); // Blank line between modules
    }

    console.log(chalk.blue(`Total modules: ${moduleGraph.nodes.size}\n`));
  }

  /**
   * Prints the graph in DOT format for visualization with Graphviz.
   */
  static exportDot(moduleGraph, options = {}) {
    const { highlightUsed = true } = options;
    let dot = 'digraph ModuleGraph {\n  node [shape=box, fontname="monospace"];\n';

    for (const [id, node] of moduleGraph.nodes) {
      const color = node.isEntry ? 'green' : (node.sideEffects ? 'orange' : 'black');
      const style = node.usedExports.size === 0 && node.exports.size > 0 ? 'dashed' : 'solid';
      dot += `  "${id}" [color="${color}", style="${style}"];\n`;

      for (const [specifier, depId] of node.dependencies) {
        dot += `  "${id}" -> "${depId}" [label="${specifier}"];\n`;
      }
    }

    dot += '}\n';
    return dot;
  }
}

module.exports = GraphLogger;
```

D. Core Logic Skeleton Code

Add debugging hooks to your main bundler pipeline:

JAVASCRIPT

```javascript
// src/bundler.js (partial)

const GraphLogger = require('./debug/graph-logger');

const DiagnosticBag = require('./diagnostics/diagnostic-bag');


class Bundler {

  constructor(options) {

    this.options = options;

    this.diagnostics = new DiagnosticBag();

    this.debug = options.debug || false;

  }


  async build() {

    try {

      // 1. Parse and extract dependencies

      const parseResults = await this.parseEntryPoints();

      if (this.debug) {

        console.log('=== After parsing ===');

        parseResults.forEach(module => {

          console.log(`${module.id}: deps = ${[...module.dependencies.keys()]}`);

        });

      }


      // 2. Resolve all modules and build graph

      const moduleGraph = await this.buildModuleGraph(parseResults);

      if (this.debug) {

        console.log('\n=== After graph building ===');

        GraphLogger.printGraph(moduleGraph, { showUnused: false });

      }


      // 3. Tree shaking

      this.markUsedExports(moduleGraph);

      if (this.debug) {

        console.log('\n=== After marking used exports ===');

        GraphLogger.printGraph(moduleGraph);

      }


      moduleGraph.shake();
```

```
      if (this.debug) {
        console.log('\n=== After shaking ===');
        GraphLogger.printGraph(moduleGraph);

        // Export DOT for visualization
        const dot = GraphLogger.exportDot(moduleGraph);
        require('fs').writeFileSync('graph.dot', dot);
        console.log('Graph exported to graph.dot (use: dot -Tpng graph.dot -o graph.png)');
      }

      // 4. Generate bundle
      const chunks = await this.generateBundle(moduleGraph);

      if (this.diagnostics.hasErrors()) {
        throw new Error('Build failed with errors');
      }

      return chunks;
    } catch (error) {
      // Convert to structured error
      this.diagnostics.add({
        code: 'BUILD_FAILED',
        message: error.message,
        severity: 'error',
        cause: error
      });
      throw error;
    }
  }

  // ... other methods
}
```

E. Language-Specific Hints

- **Use `util.inspect` for deep object logging**: `console.log(util.inspect(ast, { depth: 5, colors: true }))`.
- **Conditional debugging**: Use environment variables to control debug output: `if (process.env.DEBUG_BUNDLER) { ... }`.
- **Async stack traces**: Use `--async-stack-traces` flag in Node to get better stack traces for async operations.

- **Source map consumer**: Use the `source-map` library to parse generated source maps and query original positions: `new SourceMapConsumer(sourceMap).originalPositionFor({ line: 1, column: 10 })`.

F. Debugging Tips Table

For quick reference during implementation:

Symptom	How to Diagnose	Fix
Parser throws on valid JS	Log the raw source being parsed—check for BOM characters or unusual encoding.	Use `fs.readFile` with `utf8` encoding, strip BOM if present.
Resolution takes forever	Log each directory traversal—likely infinite loop in symlink resolution.	Implement `visitedSymlinks` set to detect cycles.
Bundle runs but exports are wrong	Compare original vs transformed AST for a single module.	Check import/export rewriting logic—ensure default vs named handling is correct.
Tree shaking removes too much	Dump graph before and after marking—see which exports aren't marked.	Ensure marking follows re-exports and namespace imports.
Source map line offsets are wrong	Use `source-map` library to verify mappings: `consumer.originalPositionFor()`.	Track line/column offsets correctly when generating code—account for wrapper lines.

Future Extensions

> **Milestone(s):** Beyond core milestones

The core bundler you've built implements the essential pipeline for transforming modular JavaScript into optimized bundles. However, real-world bundlers like webpack and Rollup offer extensive ecosystems of features that address production needs, developer experience, and specialized use cases. This section explores potential enhancements that could transform your educational project into a more production-ready tool. Each extension represents a significant but self-contained architectural addition that builds upon the foundation you've already established.

Think of your current bundler as a **basic automobile**—it reliably gets you from point A to point B with fundamental features like an engine, wheels, and steering. These extensions are like adding **climate control, a navigation system, safety features, and towing capacity**—each enhancing the vehicle's utility for specific scenarios while maintaining its core transportation function.

Extension Ideas

The following table outlines major extension areas, their primary value propositions, and the architectural components they would impact:

Extension	Primary Value	Key Components Impacted	Complexity	Learning Value
Plugin System	Extensibility without forking; community ecosystem	Entire pipeline (hooks at parse, resolve, transform, generate)	High	Design of extensible architectures, hook patterns, sandboxing
CSS/Asset Module Support	Handling non-JavaScript resources (CSS, images, fonts)	Parser (new file types), Transformer (CSS extraction), Generator (asset emission)	Medium	Resource transformation pipelines, content hashing, URL rewriting
Watch Mode & Incremental Builds	Faster development feedback loop; edit-save-refresh	CLI (file watching), Graph Builder (incremental updates), Cache (incremental results)	Medium-High	File system monitoring, incremental computation, cache invalidation
Development Server with HMR	Real-time module updates without page reload	Runtime (HMR client), Dev Server (WebSocket), Transformer (HMR code injection)	High	WebSocket communication, stateful runtime updates, module hot swapping
Code Splitting with Dynamic Imports	Lazy loading for performance optimization	Parser (dynamic import detection), Graph Builder (chunk separation), Generator (multiple bundles)	High	Async module loading, chunk dependency analysis, runtime chunk loading
On-Disk Caching	Faster rebuilds across sessions	FSCache (persistent storage), Graph Builder (cache-aware traversal), Transformer (cached transforms)	Medium	Persistent cache design, cache key generation, stale invalidation
TypeScript/Flow Integration	Type-aware bundling and checking	Parser (TypeScript AST), Transformer (type stripping), DiagnosticBag (type errors)	Medium	Type system integration, declaration file emission, type-only imports
Alternative Output Formats	Compatibility with different module systems	Generator (format-specific wrappers), Runtime (format adapters)	Low-Medium	Module system interoperability, format transpilation
Performance Profiling	Insights for optimization	DiagnosticBag (timing data), GraphLogger (visualization), CLI (profile reports)	Low	Performance measurement, bottleneck analysis, visualization

Each extension represents a meaningful architectural decision with significant implementation consequences. Below we explore the most impactful extensions in detail, complete with mental models, architectural considerations, and implementation pathways.

1. Plugin System

Mental Model: The Modular Workshop
Imagine your bundler as a **workshop with fixed assembly stations** (parse, resolve, transform, generate). A plugin system transforms it into a **modular workshop with customizable tool racks** at each station. Third-party tools (plugins) can be slotted into specific racks to modify how work gets done—sanding wood differently, applying special finishes, or even adding entirely new processing steps. The workshop's core layout remains, but its capabilities become endlessly extensible.

Architectural Impact: A plugin system fundamentally changes the bundler from a closed pipeline to an open platform. Plugins need well-defined **hook points** throughout the pipeline, a **plugin API** that exposes controlled capabilities, and a **sandboxing mechanism** to prevent plugins from breaking core functionality.

> **Decision: Rollup-Style Hook-Based Plugin Architecture**
>
> - **Context**: We need to allow external code to intercept and transform the bundling process at specific lifecycle stages while maintaining pipeline predictability.
> - **Options Considered**:
>   1. **Webpack-style loader/plugin dichotomy**: Loaders transform resources; plugins tap into compiler events. Complex but battle-tested.
>   2. **Rollup-style unified hooks**: Single plugin type with hooks for each pipeline stage. Simpler mental model.
>   3. **Middleware chain (like Vite)**: Linear sequence of transform functions. Simple but less granular control.
> - **Decision**: Adopt Rollup-style hooks for their conceptual simplicity and excellent separation of concerns.
> - **Rationale**: Rollup's hook system maps cleanly to our existing pipeline stages (resolve, load, transform, generate). Each hook has a specific purpose and expected return type, making plugin behavior predictable. The unified plugin type reduces API surface area compared to webpack's multiple plugin bases.
> - **Consequences**: Plugins can easily extend any pipeline stage, but we must carefully design hook execution order and ensure hooks don't have unintended side effects on each other. Plugin authors need clear documentation about hook timing and data availability.

Hook Points Design:

Hook Name	Stage	Purpose	Input	Expected Output
`resolveId`	Resolution	Transform module specifiers before resolution	`specifier`, `importer`	Resolved ID string or `null` to fall through
`load`	Loading	Provide module source code (can virtualize files)	`id`	`{ code: string, map?: SourceMap }` or `null`
`transform`	Transformation	Transform module AST/source after parsing	`code`, `id`	`{ code: string, map?: SourceMap }`
`moduleParsed`	Analysis	Receive parsed module info for analysis	`module` (ModuleNode)	`void` (side effects allowed)
`buildEnd`	Completion	Run after graph construction completes	`error?`	`void`
`generateBundle`	Output	Modify output chunks before writing	`chunks`, `bundle`	`void`

Common Pitfalls:

- ⚠️ **Pitfall: Uncontrolled plugin execution order**
  If plugins run in arbitrary order, transforms may conflict. **Solution**: Define explicit plugin order (first to last) and allow plugins to specify `enforce: 'pre' | 'post'` to run before/after core logic.
- ⚠️ **Pitfall: Plugins breaking source maps**
  Each transform hook should return a source map chain. **Solution**: Implement source map merging in the transformer, using libraries like `magic-string` that handle map composition automatically.
- ⚠️ **Pitfall: Plugins with infinite loops**
  A plugin calling `resolveId` on its own output could cause recursion. **Solution**: Track resolution depth and bail after a threshold (e.g., 10 levels).

Implementation Pathway:

1. Define `Plugin` interface with `name` and hooks object
2. Modify pipeline components to call hooks at appropriate points
3. Add `PluginContext` object providing limited API (e.g., `this.warn()`, `this.resolve()`)
4. Implement hook sequencing with `enforce` modifiers
5. Add plugin validation to catch common misconfigurations

2. Development Server with Hot Module Replacement

Mental Model: The Live Editing Theater

Imagine your application as a **stage play** where actors (modules) perform roles. Traditional bundling requires restarting the entire play after any

script change. HMR is like having **understudies who can swap in mid-scene**—the play continues uninterrupted while actors are replaced. The stage manager (dev server) coordinates these hot swaps via backstage signals (WebSocket), ensuring props (state) are preserved when possible.

Architectural Components:

- **Dev Server**: Serves bundled files over HTTP with WebSocket endpoint
- **HMR Runtime**: Injected into bundle, manages module hot updates
- **HMR Engine**: Tracks module relationships and update compatibility
- **File Watcher**: Detects source changes and triggers update cycles

Update Sequence:

1. File change detected → rebuild affected modules incrementally
2. Server sends update manifest over WebSocket to client
3. HMR runtime downloads updated modules (or delta patches)
4. Runtime disposes old module (calling cleanup handlers)
5. Runtime applies update and re-executes module
6. If update fails, fall back to full page reload

State Preservation Challenge: The hardest part of HMR is preserving application state across module updates. For React/Vue components, this requires framework-specific HMR integrations that understand component state semantics.

> **Decision: Webpack-style HMR with Module Hot API**
>
> - **Context**: Need a battle-tested approach that supports stateful updates for various frameworks.
> - **Options**:
>   1. **Full page reload on change**: Simple but loses application state.
>   2. **Vite-style HMR**: Leverages ES modules native `import.meta.hot`; requires browser support for dynamic import.
>   3. **Webpack HMR**: Custom `module.hot` API; works in all browsers, supports state preservation.
> - **Decision**: Implement webpack-style `module.hot` API for maximum compatibility.
> - **Rationale**: While Vite's approach is more elegant, it requires modern browsers and doesn't handle state preservation as robustly for arbitrary modules. Webpack's API provides explicit hooks (`accept`, `dispose`) that give developers control over update behavior.
> - **Consequences**: Larger runtime footprint (~3-5KB), more complex implementation, but works reliably across all browsers and supports complex update scenarios.

HMR Runtime API:

```javascript
// In bundled modules
if (module.hot) {
  module.hot.accept(); // Accept updates to this module
  module.hot.dispose((data) => {
    // Cleanup before module is replaced
    data.someState = currentState;
  });
  module.hot.accept('./dep.js', () => {
    // Custom handler when dependency updates
    reinitializeWithNewDep();
  });
}
```

Implementation Steps:

1. Extend runtime to include HMR client code
2. Add WebSocket server to dev server
3. Implement incremental rebuild for changed files
4. Generate update payloads (added/changed/removed modules)
5. Add HMR API to module wrapper (`module.hot` object)
6. Implement module acceptance and disposal logic

3. Code Splitting with Dynamic Imports

Mental Model: The Modular Library with On-Demand Sections

Imagine a **physical library** where all books are bound into a single massive volume. Code splitting transforms it into a **main reference desk with pull-out specialty sections**—you only fetch the specialized book when needed. Dynamic imports (`import()`) are the **call slips** requesting specific sections, while the runtime becomes the **librarian** who knows where each section is shelved and can fetch it on demand.

Architectural Changes Required:

- **Chunk Graph**: Extends `ModuleGraph` to group modules into `BundleChunk` objects based on entry points and dynamic import boundaries
- **Async Dependency Tracking**: `ModuleNode` needs to track which dynamic imports lead to which chunks
- **Runtime Chunk Loader**: Enhanced runtime that can load chunks on demand
- **Chunk Manifest**: Mapping from module IDs to chunk files for runtime resolution

Chunk Splitting Strategies:

Strategy	Description	Use Case
Entry Point Splitting	Separate chunk per entry point	Multi-page applications
Dynamic Import Splitting	New chunk for each `import()`	Route-based code splitting
Vendor Splitting	Separate chunk for `node_modules`	Better caching of third-party code
Common Chunk Extraction	Shared modules moved to common chunk	Avoid duplication across chunks

> **Decision: Deterministic Chunk Hashing for Long-Term Caching**
>
> - **Context**: Need to generate stable chunk filenames for browser caching while ensuring changes bust the cache.
> - **Options**:
>   1. **Incremental numbers**: `0.js` , `1.js` (unstable across builds)
>   2. **Content hash**: `chunk-abc123.js` (stable if content unchanged)
>   3. **Manifest with mapping**: Separate manifest file maps module IDs to chunk names
> - **Decision**: Use content hash in chunk filenames (`[name].[contenthash:8].js`) with a runtime manifest.
> - **Rationale**: Content hashing provides optimal caching—unchanged chunks keep the same filename across builds. The runtime manifest (small file) maps module IDs to chunk names and can be loaded initially.
> - **Consequences**: More complex build output (multiple files), need to manage chunk dependencies (load B before C), and potential for "hash churn" if minor changes affect many chunks.

Chunk Graph Algorithm:

1. Identify **entry chunks** from configured entry points
2. For each dynamic import, create **async boundary** and new chunk
3. Perform **reachability analysis** to assign modules to chunks
4. **Optimize chunk assignments** to minimize duplication and requests
5. **Topologically sort chunks** for dependency ordering

Common Pitfalls:

- ⚠️ **Pitfall: Circular chunk dependencies**
  Chunk A needs B, B needs A. **Solution**: Detect cycles and merge chunks or implement async dependency resolution.

- ⚠️ **Pitfall: Duplicate modules across chunks**
  Same module ends up in multiple chunks, increasing bundle size. **Solution**: Implement deduplication via shared chunks or module federation.
- ⚠️ **Pitfall: Missing chunk preloading**
  Browser discovers dynamic imports too late. **Solution**: Generate `<link rel="preload">` hints or use `import()` with webpack magic comments.

4. CSS and Asset Module Support

Mental Model: The Multilingual Document Processor

Your bundler currently understands only **JavaScript grammar**. Adding CSS/asset support is like hiring **specialist translators** for different languages—they extract meaning from foreign syntax and convert it to something the main system can process. CSS modules become **translated style objects**, images become **URL references or inline data**, all while maintaining the original intent.

Processing Pipeline Extension:

```
CSS File → CSS Parser (postcss) → CSS Transformer (autoprefixer) → CSS Extractor (separate .css file or JS exports)
Image File → Image Transformer (optimize) → Asset Emitter (file copy) → URL Generator (hashed path)
```

CSS Processing Strategies:

Strategy	Output	Pros	Cons
Extracted CSS	Separate `.css` file with `<link>`	Leverages browser CSS pipeline, cacheable	FOUC risk, extra HTTP request
CSS-in-JS	Styles injected via JavaScript	No FOUC, critical CSS extraction	Larger JS bundle, slower rendering
CSS Modules	Scoped class names as JS object	Local scoping, dead CSS elimination	Build-time only, runtime overhead

> **Decision: Extracted CSS with CSS Modules Support**
>
> - **Context**: Need to handle CSS effectively while supporting modern component-scoped styling.
> - **Options**:
>   1. **CSS extraction only**: Traditional separate CSS files.
>   2. **CSS-in-JS only**: All styles in JavaScript bundles.
>   3. **Hybrid approach**: Extract global CSS, support CSS Modules for components.
> - **Decision**: Implement extracted CSS as default with opt-in CSS Modules support.
> - **Rationale**: Extracted CSS leverages browser's parallel CSS loading and caching. CSS Modules provide the scoping benefits of CSS-in-JS without runtime overhead. This covers the majority of real-world use cases.
> - **Consequences**: Need to implement CSS parser, extractor, and URL rewriting for assets referenced in CSS. CSS Modules require generating and injecting scoped class name mappings.

Asset Handling Architecture:

- **Asset Registry**: Tracks all non-JS resources and their output paths
- **URL Rewriter**: Transforms `url()` references in CSS to final output paths
- **Content Hasher**: Generates hash-based filenames for cache busting
- **Asset Emitter**: Writes assets to output directory

Implementation Sequence:

1. Add `asset` module type to `ModuleNode` (distinct from `js`)
2. Extend parser to recognize CSS files (`.css`, `.scss` with plugin)
3. Implement CSS transformer pipeline (postcss with plugins)
4. Add asset loading hooks to module resolution
5. Implement CSS extractor that creates separate `BundleChunk` for CSS
6. Add URL rewriting for assets referenced in CSS/JS

Implementation Guidance

While implementing all extensions would be a massive undertaking, here's a pragmatic roadmap starting with the highest-value additions:

Technology Recommendations:

Extension	Core Dependency	Complementary Tools
Plugin System	Native events/hooks	`tapable` (webpack's plugin engine) for advanced hook management
Dev Server	`express` + `ws` (WebSocket)	`chokidar` for file watching, `sirv` for static serving
Code Splitting	Native `import()` parsing	`magic-string` for source map handling during chunk separation
CSS Support	`postcss` for CSS processing	`cssnano` for minification, `postcss-modules` for CSS Modules
TypeScript	`typescript` compiler API	`@babel/preset-typescript` for lighter-weight transpilation

Recommended Extension Implementation Order:

1. **Plugin System** (enables all other extensions via plugins)
2. **Watch Mode** (foundation for dev server)
3. **CSS Support** (high utility, moderate complexity)
4. **Dev Server** (requires watch mode and plugin system)
5. **Code Splitting** (most complex, but transformative)

File Structure for Extended Bundler:

```
bundler/
  core/                   # Existing core pipeline
    parser/
    resolver/
    graph-builder/
    generator/
  extensions/             # New: optional extensions
    plugins/
      plugin-api.js       # Plugin interface and hook registry
      builtin-plugins/    # CSS, assets, etc. as builtin plugins
    dev-server/
      server.js           # HTTP + WebSocket server
      hmr-engine.js       # HMR update logic
      client/             # HMR client runtime code
    code-splitting/
      chunk-graph.js      # Extends ModuleGraph for chunks
      chunk-manifest.js   # Runtime chunk mapping
    assets/
      css-processor.js
      asset-registry.js
  cli.js                  # Extended with --watch, --serve flags
```

Plugin System Starter Code:

JAVASCRIPT

```javascript
// extensions/plugins/plugin-api.js

class PluginContext {

  constructor(bundler, plugin) {

    this.bundler = bundler;

    this.plugin = plugin;

  }



  warn(message, location) {

    this.bundler.diagnostics.add({

      code: 'PLUGIN_WARNING',

      message: `[${this.plugin.name}] ${message}`,

      severity: 'warning',

      location

    });

  }



  async resolve(specifier, importer) {

    return this.bundler.resolver.resolve(specifier, importer);

  }

}


class HookRegistry {

  constructor() {

    this.hooks = new Map();

  }



  registerHook(name, options = {}) {

    const hook = new AsyncSeriesHook(options); // Using tapable pattern

    this.hooks.set(name, hook);

    return hook;

  }



  async callHook(name, context, ...args) {

    const hook = this.hooks.get(name);

    if (!hook) return null;

    return await hook.promise(context, ...args);

  }
```

```
}

// Usage in core components:
class GraphBuilder {
  constructor(options) {
    this.pluginSystem = options.pluginSystem;
  }

  async build(entryPaths) {
    // Call resolveId hook for each module
    const resolvedId = await this.pluginSystem.callHook(
      'resolveId',
      pluginContext,
      specifier,
      importer
    );

    // If hook returned a value, use it; otherwise proceed with normal resolution
    // ... rest of build logic
  }
}
```

Watch Mode Implementation Pattern:

```javascript
// extensions/watch-mode/watcher.js

const chokidar = require('chokidar');


class IncrementalBuilder {

  constructor(bundler) {

    this.bundler = bundler;

    this.cache = new Map(); // Module hash → output cache

    this.watcher = null;

  }



  async watch(entryPoints, options) {

    // 1. Initial full build

    const graph = await this.bundler.graphBuilder.build(entryPoints);

    const chunks = await this.bundler.generateBundle(graph, options);



    // 2. Set up file watcher

    this.watcher = chokidar.watch([...graph.nodes.keys()], {

      ignored: /node_modules/,

      persistent: true

    });



    // 3. On change, rebuild affected modules

    this.watcher.on('change', async (filePath) => {

      console.log(`File changed: ${filePath}`);



      // TODO: Determine affected module and its dependents

      // TODO: Rebuild only those modules

      // TODO: Update cache with new results

      // TODO: Emit updated chunks

    });

  }

}
```

Milestone Checkpoints for Extensions:

1. **After Plugin System**:

```
   $ node bundler.js --plugin ./my-plugin.js entry.js
   # Should see "MyPlugin: transform hook called for entry.js"
   # Bundle should include plugin transformations
```

2. **After Watch Mode**:

```
$ node bundler.js --watch entry.js
# Initial build completes
# Modify entry.js → should automatically rebuild in <500ms
# Bundle file timestamp updates
```

3. **After CSS Support**:

```
$ node bundler.js --entry ./styles.css
# Should output .css file with processed CSS
# Should handle @import and url() references correctly
```

4. **After Dev Server**:

```
$ node bundler.js --serve --port 3000 entry.js
# Server starts at http://localhost:3000
# Browser loads page with HMR runtime
# Edit file → browser updates without refresh
```

Debugging Extension Issues:

Symptom	Likely Cause	Diagnosis	Fix
Plugin hooks not firing	Hook not registered or called	Check hook registration in pipeline stage	Ensure hook is called with correct arguments
HMR updates cause state loss	Module not accepting updates	Check for `module.hot.accept()`	Add accept handler or preserve state in `dispose`
Chunks loading in wrong order	Missing chunk dependency	Inspect chunk manifest dependencies	Ensure parent chunks reference child chunks in manifest
CSS URLs broken in production	Incorrect public path	Check asset URLs in final CSS	Configure `publicPath` option for asset resolution

These extensions represent natural evolution points for your bundler. Each addresses real production needs while teaching advanced architectural patterns. Start with the plugin system—it's the gateway to all other extensions and teaches the invaluable skill of designing extensible systems.