# Development Specification 2

[**Development Specification 2 1**](#_7c91kvt2ektz)

[1. Header 1](#_1yx7gv1wkqw7)

[2. Architecture Diagram 2](#_vfi2uqi2c3g0)

[3. Class Diagram 3](#_r98hm4ln20xj)

[4. List of Classes 5](#_mpj1trj360a4)

[5. State Diagrams 7](#_bzqj4iqw922v)

[6. Flow Chart 8](#_e8ukzq81cqwy)

[7. Development Risks and Failures 10](#_y39r6888ynhu)

[8. Technology Stack 11](#_pc4sta61msk9)

[9. APIs 13](#_2zwm0f39ov7h)

[10. Public Interfaces 15](#_90gpobphl8ig)

[11. Data Schemas 18](#_vfd1gecep9mi)

[12. Reflection 19](#_nj7vnr3x61yv)

**User story:**

As a developer, I want the AI to generate multiple related code files at once so that I do not need to manually stitch files together.

**GitHub Link**:

### *1. Header*

**Version**: v0.1 (2025-09-23)  
**Authors & Roles**:

* [Baolin Shi](mailto:baolins@andrew.cmu.edu)– Initial draft for the dev spec
* [Christina Ma](mailto:qianoum@andrew.cmu.edu) - validate the initial draft
* [Dejun Yang](mailto:dejuny@andrew.cmu.edu) - validate the initial draft

#### ***Chat Log Link****:* [*https://chatgpt.com/share/68d43a87-a7d0-8008-a783-597f967b808f*](https://chatgpt.com/share/68d43a87-a7d0-8008-a783-597f967b808f)

### *2. Architecture Diagram*

#### 

#### **Where Components Run**

#### IDE Client (VS Code Extension, local) US2Controller, DiffPreviewer, WorkspaceFSAdapter, RollbackManager, ProjectContextProvider, LLMClient ↳ Runs inside VS Code on the developer’s machine. Backend Service (local or remote microservice) PromptManager, ArtifactPlanner, ScaffoldGenerator, CrossFileCoordinator, LanguageCheckers ↳ Handles prompt normalization, planning, file scaffolding, consistency, and validation.

#### Cloud LLM (remote provider) LargeLanguageModel ↳ Hosted by a third-party API (e.g., OpenAI, Anthropic). Processes normalized prompts and returns candidate file drafts.

#### **Information Flows**

#### User Input → US2Controller (IDE)

#### US2Controller → ProjectContextProvider (collect repo info)

#### US2Controller → LLMClient (sends prompt + context)

#### LLMClient → PromptManager → ArtifactPlanner (create ChangePlan)

#### ArtifactPlanner → ScaffoldGenerator → LargeLanguageModel (generate drafts)

#### CrossFileCoordinator ensures imports/types consistent across drafts

#### LanguageCheckers validates syntactic correctness, produces Diagnostics

#### Backend returns validated ArtifactSet → LLMClient → US2Controller

#### DiffPreviewer shows changes; WorkspaceFSAdapter writes them; RollbackManager snapshots state.

#### ***Rationale****:*

This architecture cleanly separates orchestration (IDE extension), planning + validation (backend service), and text generation (cloud LLM).

* Keeping planning and consistency checks outside the LLM reduces hallucination risks.
* The IDE extension handles user control (preview, accept/reject, rollback).
* Backend services ensure that generated artifacts are consistent and validated before they touch the workspace.
* Cloud LLM is used only for free-form generation (code drafts).

This separation makes the system modular, testable, and resilient against failures in any one layer.

#### ***Chat Log Link****:* [*https://www.mermaidchart.com/d/d63210f1-cfd8-462d-a34a-616ed63c30dd*](https://www.mermaidchart.com/d/d63210f1-cfd8-462d-a34a-616ed63c30dd)

#### 

### *3. Class Diagram*

#### 

#### ***Rationale****:*

We designed the class diagram to clearly show the main roles in multi-file code generation and how they connect.

1. Abstract classes at the top

* We added base classes (like BaseController, BaseRetriever, BaseValidator) so common ideas are grouped.
* Concrete classes (like US2Controller, ScaffoldGenerator, LanguageCheckers) are at the bottom, following the guideline of top-to-bottom hierarchy.

1. Consistency with the architecture diagram

* All classes here match what we already showed in the architecture diagram.
* This keeps the spec consistent across diagrams.

1. Fields and methods are minimal

* We only included fields and methods directly needed for the user story.
* The flow of methods follows the user steps: plan → generate → validate → preview → apply/rollback.

1. Data types are leaves

* ChangePlan, ArtifactSet, Snapshot, and Diagnostic are just data containers, so we only show fields.

1. Use of inheritance vs composition

* Inheritance is used where behavior is shared (like all validators having validate()).
* Composition is used where one class holds another (like snapshots inside the workspace manager).

Overall, the diagram keeps things simple, avoids duplication, and makes the flow from abstract roles to concrete classes easy to see.

#### ***Chat Log Link****:*

#### [***https://www.mermaidchart.com/d/beef52d7-7b27-4d5f-a845-b315d67a5cd1***](https://www.mermaidchart.com/d/beef52d7-7b27-4d5f-a845-b315d67a5cd1)

### *4. List of Classes*

#### **IDE Client Module**

#### CTRL2.1 – US2Controller

#### Orchestrates the multi-file code generation workflow. Starts generation, confirms change plans, and applies artifacts to the workspace.

#### CTRL2.2 – DiffPreviewer

#### Provides a preview of generated file changes and lets the user accept or reject them individually or in bulk.

#### CTRL2.3 – WorkspaceFSAdapter

#### Interfaces with the local VS Code file system. Applies generated artifacts, takes workspace snapshots, and supports rollback.

#### CTRL2.4 – RollbackManager

#### Manages workspace snapshots for safe rollback if multi-file code generation results are incorrect or undesired.

#### CTRL2.5 – ProjectContextProvider

#### Provides project-level context (workspace root, language detection, path resolution) to inform artifact planning and file generation.

#### CTRL2.6 – LLMClient

#### Communicates with the Copilot backend to request artifact planning and file generation based on the user’s prompt and context.

#### **Backend Service Module**

#### SVC2.1 – PromptManager

#### Normalizes and structures the user’s input prompt before sending it to the LLM.

#### SVC2.2 – ArtifactPlanner

#### Produces a plan of related files to be generated, including their roles and relationships.

#### SVC2.3 – ScaffoldGenerator

#### Creates file drafts from the change plan and ensures generated content is structured for the repo.

#### SVC2.4 – CrossFileCoordinator

#### Ensures consistency across multiple generated files (e.g., imports, references).

#### SVC2.5 – LanguageCheckers

#### Validates generated artifacts against language specifications (syntax and style).

#### **Cloud LLM Module**

#### CLOUD2.1 – LargeLanguageModel

#### Cloud-hosted model that generates draft code files and responses when provided with normalized prompts.

#### **Data Storage Classes (Structs)**

#### DT2.1 – ChangePlan

#### Represents the planned set of files to be generated and a summary of the proposed changes.

#### DT2.2 – ArtifactSet

#### Holds the actual generated artifacts (multi-file drafts) before they are applied.

#### DT2.3 – Snapshot

#### Stores metadata about a workspace snapshot for rollback purposes.

#### DT2.4 – Diagnostic

#### Represents issues detected in generated code, such as errors, warnings, or inconsistencies.

#### ***Rationale****:*

We grouped the classes by Client, Backend, Cloud, and Data Storage to align with the architecture diagram and highlight the separation of responsibilities. Each class label (e.g., CTRL2.1) makes it easy to trace features across the spec. Only classes directly relevant to multi-file code generation are included, while supporting data structures are separated out for clarity. This approach ensures consistency between the architecture diagram and the class list, while keeping the design easy to follow and extend.

### *5. State Diagrams*

#### 

#### ***Rationale****:*

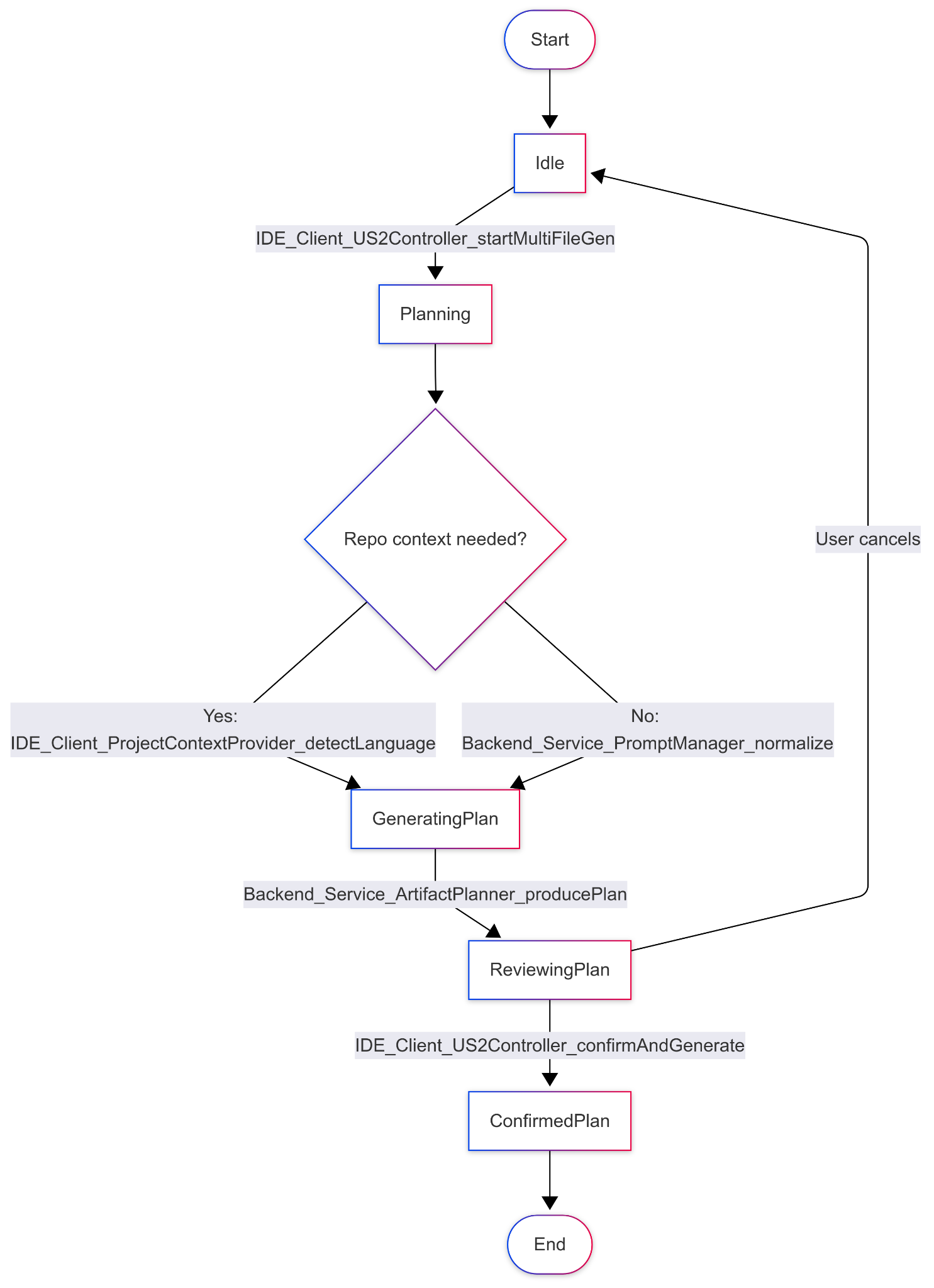
The state diagram captures the full lifecycle of multi-file code generation, from the user’s initial prompt to applying artifacts into the workspace. We included all critical transitions: planning, draft generation, validation, preview, and final application. Rollback is modeled as an explicit error-handling path to ensure safety when generation fails or the user rejects changes. Each state change is tied to specific class methods (e.g., US2Controller.startMultiFileGen(), ScaffoldGenerator.generateFileDrafts()), keeping the diagram consistent with the architecture and class diagrams. This provides a clear blueprint of how data fields (activePlan, pendingArtifacts, diagnostics, etc.) evolve as the system progresses.

#### ***Chat Log Link****:*

<https://www.mermaidchart.com/d/85c73712-8b24-4bfd-9cb0-0a1e1030946c>

### *6. Flow Chart*

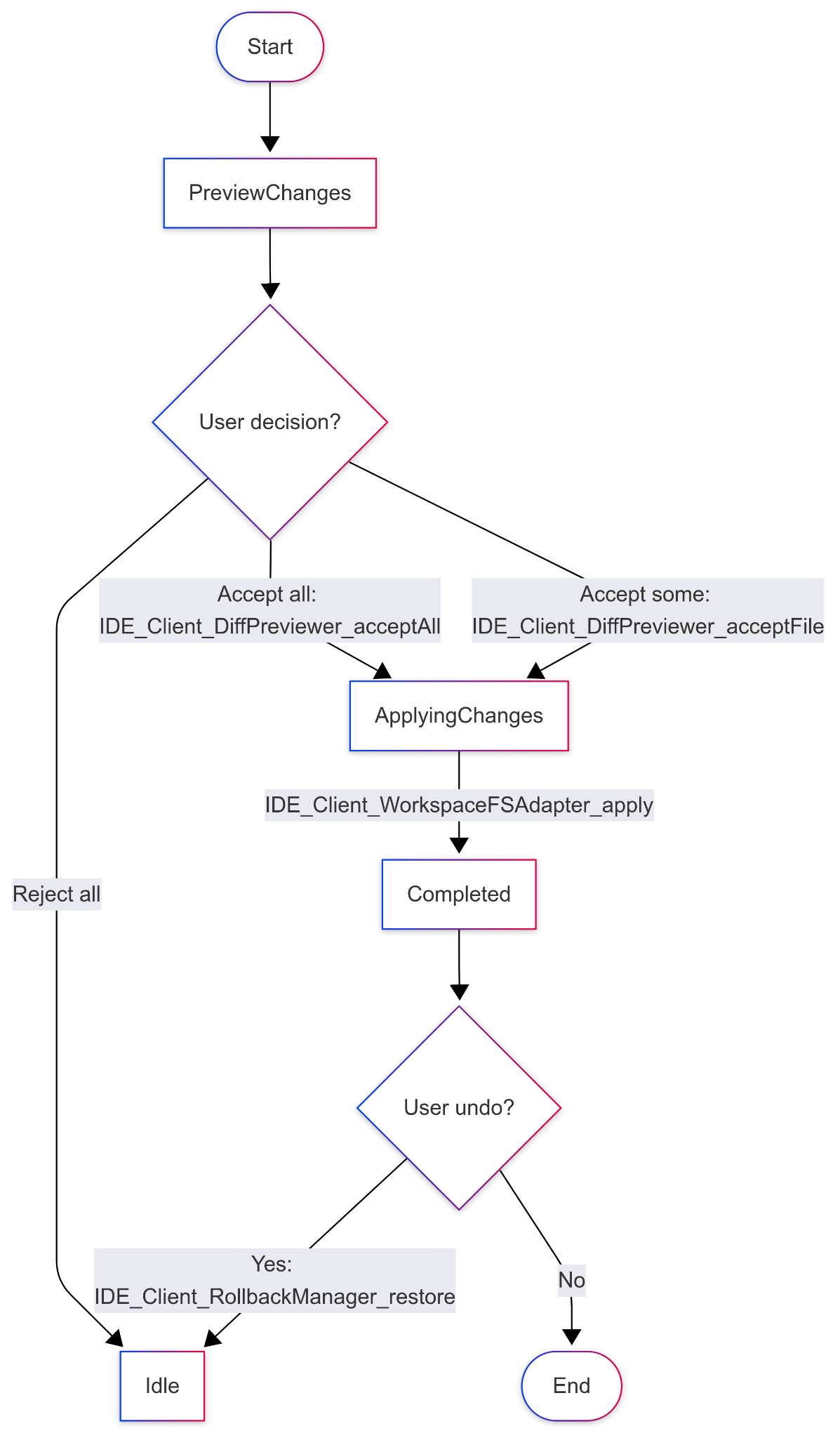
**Scenario 1: Planning a Multi-file Generation Request (US2.SC1)**

****

**Scenario 2: US2.SC2 — Generate drafts and validate**

#### 

**Scenario 3: US2.SC3 — Preview, apply, and rollback**



#### ***Rationale****:*

We created three separate flow charts to capture the key scenarios of User Story 2: planning a multi-file generation request, generating/validating drafts, and previewing/applying results. Each chart maps directly to states already defined in the State Diagram, so the two notations stay consistent. Method calls are shown as edge labels to highlight the system actions that trigger transitions. To avoid Mermaid parsing errors, we used simplified underscore labels in the diagrams and provided a legend that maps them back to the full method names. This keeps the flow charts readable, technically accurate, and aligned with the architecture and class diagrams.

### 

### *7. Development Risks and Failures*

**Runtime Failures**

* R2.1 – Crash in US2Controller
  + Effect: User request aborted, no files generated, UI frozen.
  + Recovery: Restart extension; request reissued.
* R2.2 – WorkspaceFSAdapter loses runtime state
  + Effect: File system context lost; generation cannot apply.
  + Recovery: Reload workspace; regenerate artifacts.
* R2.3 – RollbackManager snapshot corrupted
  + Effect: Undo operation fails; risk of unrecoverable changes.
* R2.4 – LLMClient RPC failure
  + Effect: Plan or generation request times out.
  + Recovery: Retry with exponential backoff; surface error to user.
* R2.5 – Backend server overloaded / out of RAM
  + Effect: Slow or failed artifact planning; degraded response times.
  + Recovery: Queue requests, auto-scale backend nodes.

**Connectivity Failures**

* C2.1 – Lost connectivity to LLM service
  + Effect: User cannot generate files; plan request fails.
  + Recovery: Cache request; retry after reconnect.
* C2.2 – Network traffic spikes
  + Effect: Latency increases; user perceives slowdown.
  + Recovery: Use circuit breaker and retry policy.

**Hardware Failures**

* H2.1 – Backend server crash
  + Effect: Planning/validation unavailable.
  + Recovery: Failover to redundant instance.
* H2.2 – Bad configuration loaded
  + Effect: Wrong language check rules or scaffold templates.
  + Recovery: Rollback to last known good config.

**Intruder / Security Risks**

* I2.1 – Denial of Service attack on backend
  + Effect: All multi-file generations stall; user-facing downtime.
  + Recovery: Throttle requests; enable rate limiting.
* I2.2 – Unauthorized access to workspace snapshot
  + Effect: Malicious edits injected.
  + Recovery: Restrict access with VS Code API permissions.
* I2.3 – Tampered response from LLM
  + Effect: Malicious or misleading generated code.
  + Recovery: Add validation & guardrails; prompt sanitization.
* I2.4 – HTTP session hijacked (LLMClient)
  + Effect: Attacker gains access to prompt/response stream.
  + Recovery: Enforce TLS, rotate API keys.

**Ranking of Failures (Likelihood vs. Impact)**

1. Most Likely: RPC failures and connectivity loss (frequent in network-heavy workflows).
2. Highest Impact: Rollback corruption (user cannot recover), tampered LLM responses (security risk).
3. Moderate: Hardware server crashes and DoS (less likely but service-wide impact).
4. Low: Workspace snapshot corruption (rare with standard FS APIs).

#### ***Rationale****:*

We analyzed risks by module to capture both user-visible (UI errors, failed generations) and internally-visible effects (loss of state, backend overload). Each risk is uniquely labeled, with clear recovery procedures (restart, rollback, retry, failover). Security and privacy risks (e.g., tampered responses, hijacked sessions) are included to account for the cloud dependency. This ensures the spec addresses both common runtime errors and less frequent but high-impact failures.

### 

### *8. Technology Stack*

**IDE / Client Side**

* TECH2.1 – Visual Studio Code (v1.92 or later)
  + Use: IDE environment where developers request multi-file generation. Provides file system APIs, extension hooks, and UI integration.
  + Why chosen: Industry-standard editor with extensibility and existing Copilot integration.
  + URL: <https://code.visualstudio.com/>
* TECH2.2 – TypeScript (v5.3)
  + Use: Primary implementation language for VS Code extension logic (US2Controller, DiffPreviewer, WorkspaceFSAdapter).
  + Why chosen: Strong typing improves maintainability and aligns with VS Code extension ecosystem.
  + URL: <https://www.typescriptlang.org/>
* TECH2.3 – Node.js (v20 LTS)
  + Use: Runtime for VS Code extension and backend service stubs.
  + Why chosen: Widely supported for extension development and integrates easily with npm ecosystem.
  + URL: https://nodejs.org/

**Backend Service**

* TECH2.4 – Express.js (v4.x)
  + Use: Lightweight web server for routing artifact planning requests, prompt normalization, and validation tasks.
  + Why chosen: Simplicity, wide adoption, and compatibility with Node.js.
  + URL: <https://expressjs.com/>
* TECH2.5 – Docker (v24)
  + Use: Containerization of backend microservices (PromptManager, ArtifactPlanner, ScaffoldGenerator).
  + Why chosen: Ensures reproducible deployment across developer and CI/CD environments.
  + URL: https://www.docker.com/

**Cloud LLM Integration**

* TECH2.6 – OpenAI GPT-4 API (2025 version)
  + Use: Generates file drafts and artifact plans from normalized prompts.
  + Why chosen: Proven accuracy and support for structured multi-file responses.
  + URL: <https://platform.openai.com/docs/>
* TECH2.7 – REST/JSON over HTTPS
  + Use: Communication protocol between VS Code client, backend services, and LLM API.
  + Why chosen: Standard, secure, interoperable.
  + URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

**Supporting Tools**

* TECH2.8 – ESLint (v8.x)
  + Use: Static analysis and style enforcement for extension code.
  + Why chosen: Improves code quality and consistency.
  + URL: <https://eslint.org/>
* TECH2.9 – Jest (v29.x)
  + Use: Unit testing framework for controllers, planners, and validators.
  + Why chosen: Strong ecosystem and TypeScript support.
  + URL: https://jestjs.io/

#### ***Rationale****:*

The chosen stack prioritizes developer familiarity (TypeScript, Node.js, VS Code), deployment reproducibility (Docker), and scalability with LLM integration (OpenAI API). These technologies are widely supported, lowering risk while ensuring the system can be realistically prototyped and extended.

### 

### *9. APIs*

**IDE Client Module**

CTRL2.1 – US2Controller

+ startMultiFileGen(prompt: string, options: GenOptions): void (public)

+ confirmAndGenerate(plan: ChangePlan): void (public)

+ applyArtifacts(artifacts: ArtifactSet): Promise (public)

Role: Orchestrates the overall multi-file workflow.

CTRL2.2 – DiffPreviewer

+ render(artifacts: ArtifactSet): void (public)

+ acceptAll(): void (public)

+ acceptFile(path: string): void (public)

Role: Renders and manages preview of generated artifacts.

CTRL2.3 – WorkspaceFSAdapter

+ snapshot(): SnapshotId (public)

+ apply(artifacts: ArtifactSet, selection?: string[]): Promise (public)

+ rollback(snapshot: SnapshotId): Promise (public)

Role: Manages workspace filesystem state.

CTRL2.4 – RollbackManager

+ createSnapshot(): SnapshotId (public)

+ restore(id: SnapshotId): Promise (public)

Role: Provides safe rollback of workspace changes.

CTRL2.5 – ProjectContextProvider

+ detectLanguage(): LanguageSpec (public)

+ resolvePaths(request: GenRequest): PathHints (public)

Role: Supplies project-level metadata for planning and generation.

CTRL2.6 – LLMClient

+ planArtifacts(prompt: string, ctx: Context): Promise (public)

+ generateFiles(plan: ChangePlan, ctx: Context): Promise (public)

Role: Handles communication with backend/LLM.

**Backend Service Module**

SVC2.1 – PromptManager

+ normalize(prompt: string, ctx: Context): NormalizedPrompt (public)

SVC2.2 – ArtifactPlanner

+ producePlan(prompt: NormalizedPrompt): ChangePlan (public)

SVC2.3 – ScaffoldGenerator

+ generateFileDrafts(plan: ChangePlan): DraftSet (public)

SVC2.4 – CrossFileCoordinator

+ ensureConsistency(drafts: DraftSet): DraftSet (public)

SVC2.5 – LanguageCheckers

+ validate(artifacts: ArtifactSet, lang: LanguageSpec): Diagnostic[] (public)

**Cloud LLM Module**

CLOUD2.1 – LargeLanguageModel

+ generateResponse(prompt: NormalizedPrompt): DraftSet (public)

**Data Types (Structs)**

(Fields only, no methods — grouped for completeness)

* DT2.1 – ChangePlan → { files: PlannedFile[], summary: string }
* DT2.2 – ArtifactSet → { artifacts: Artifact[] }
* DT2.3 – Snapshot → { id: string, time: Date, fileCount: int }
* DT2.4 – Diagnostic → { path: string, severity: string, message: string }

#### ***Rationale****:*

We listed all methods grouped by class and module to match the architecture diagram. Return types and parameter types clarify how components interact, while labels (e.g., CTRL2.1) provide traceability across spec sections. Private methods were excluded since they are not part of the public interface, keeping the API focused on developer-visible integration points.

### *10. Public Interfaces*

**IDE Client Module**

CTRL2.1 – US2Controller

* Public Methods:
  + startMultiFileGen(prompt: string, options: GenOptions): void
  + confirmAndGenerate(plan: ChangePlan): void
  + applyArtifacts(artifacts: ArtifactSet): Promise<void>
* Uses from Other Components:
  + Calls ProjectContextProvider.detectLanguage() (CTRL2.5)
  + Calls LLMClient.planArtifacts() and generateFiles() (CTRL2.6)
  + Passes results to DiffPreviewer.render() (CTRL2.2)
  + Sends confirmed artifacts to WorkspaceFSAdapter.apply() (CTRL2.3)

CTRL2.2 – DiffPreviewer

* Public Methods:
  + render(artifacts: ArtifactSet): void
  + acceptAll(): void
  + acceptFile(path: string): void
* Uses:
  + Receives artifacts from US2Controller.

CTRL2.3 – WorkspaceFSAdapter

* Public Methods:
  + snapshot(): SnapshotId
  + apply(artifacts: ArtifactSet, selection?: string[]): Promise<void>
  + rollback(snapshot: SnapshotId): Promise<void>
* Uses:
  + Depends on RollbackManager.restore() for rollback.

CTRL2.4 – RollbackManager

* Public Methods:
  + createSnapshot(): SnapshotId
  + restore(id: SnapshotId): Promise<void>
* Uses:
  + Called by WorkspaceFSAdapter to manage safe undo operations.

CTRL2.5 – ProjectContextProvider

* Public Methods:
  + detectLanguage(): LanguageSpec
  + resolvePaths(request: GenRequest): PathHints
* Uses:
  + Provides context to US2Controller and indirectly to LLMClient.

CTRL2.6 – LLMClient

* Public Methods:
  + planArtifacts(prompt: string, ctx: Context): Promise<ChangePlan>
  + generateFiles(plan: ChangePlan, ctx: Context): Promise<ArtifactSet>
* Uses from Backend Module:
  + Calls PromptManager.normalize() (SVC2.1)
  + Calls ArtifactPlanner.producePlan() (SVC2.2)
  + Calls ScaffoldGenerator.generateFileDrafts() (SVC2.3)

**Backend Service Module**

SVC2.1 – PromptManager

* Public Method:
* normalize(prompt: string, ctx: Context): NormalizedPrompt
* Used by: LLMClient

SVC2.2 – ArtifactPlanner

* Public Method:
* producePlan(prompt: NormalizedPrompt): ChangePlan
* Used by: LLMClient

SVC2.3 – ScaffoldGenerator

* Public Method:
* generateFileDrafts(plan: ChangePlan): DraftSet
* Uses:
  + Calls CrossFileCoordinator.ensureConsistency() (SVC2.4)
  + Calls LanguageCheckers.validate() (SVC2.5)
  + Calls LargeLanguageModel.generateResponse() (CLOUD2.1)

SVC2.4 – CrossFileCoordinator

* Public Method:
  + ensureConsistency(drafts: DraftSet): DraftSet
  + Used by: ScaffoldGenerator

SVC2.5 – LanguageCheckers

* Public Method:
  + validate(artifacts: ArtifactSet, lang: LanguageSpec): Diagnostic[]
  + Used by: ScaffoldGenerator

**Cloud LLM Module**

CLOUD2.1 – LargeLanguageModel

* Public Method:
  + generateResponse(prompt: NormalizedPrompt): DraftSet
  + Used by: ScaffoldGenerator

#### ***Rationale****:*

This section highlights only the public-facing methods that developers or components rely on, filtering out private/internal details. Interfaces are grouped by module, with clear notes on which methods are consumed by other components or modules. This mapping ensures traceability between modules (client → backend → cloud) and clarifies the integration boundaries for multi-file code generation.

### *11. Data Schemas*

#### DT2.1 – ChangePlan

#### Held by Classes: US2Controller, ArtifactPlanner

#### Database Table: change\_plans

#### Columns:

#### plan\_id → UUID (16 bytes)

#### summary → TEXT (variable, avg 256 bytes)

#### files → JSONB (1–5 KB depending on file count)

#### Notes: files stored as JSON array for flexibility across languages.

#### Estimated Storage: ~2–6 KB per plan.

#### DT2.2 – ArtifactSet

#### Held by Classes: US2Controller, DiffPreviewer, WorkspaceFSAdapter

#### Database Table: artifact\_sets

#### Columns:

#### artifact\_set\_id → UUID (16 bytes)

#### plan\_id → UUID (foreign key to change\_plans)

#### artifacts → JSONB (stores array of code artifacts)

#### Estimated Storage: 10 KB to 500 KB depending on number/size of generated files.

#### DT2.3 – Snapshot

#### Held by Classes: RollbackManager, WorkspaceFSAdapter

#### Database Table: snapshots

#### Columns:

#### snapshot\_id → UUID (16 bytes)

#### created\_at → TIMESTAMP (8 bytes)

#### file\_count → INT (4 bytes)

#### metadata → JSONB (avg 1 KB)

#### Estimated Storage: ~1–2 KB per snapshot.

#### DT2.4 – Diagnostic

#### Held by Classes: LanguageCheckers

#### Database Table: diagnostics

#### Columns:

#### diagnostic\_id → UUID (16 bytes)

#### artifact\_set\_id → UUID (foreign key to artifact\_sets)

#### path → VARCHAR(255) (~255 bytes)

#### severity → VARCHAR(16) (avg 16 bytes)

#### message → TEXT (avg 512 bytes)

#### Estimated Storage: ~1 KB per diagnostic entry.

#### ***Rationale****:*

We separated data types into normalized tables to reflect their different lifecycles: ChangePlans describe intended changes, ArtifactSets store generated results, Snapshots allow rollback, and Diagnostics record validation issues. JSONB was chosen for flexible storage of structured fields (e.g., file lists, artifact content) while still allowing indexing in PostgreSQL. This design balances extensibility (handling variable file structures) with performance.

### *12. Reflection*

Using the LLM to create my development specifications was very helpful, especially in breaking down vague ideas into structured diagrams, class lists, and risk analyses. The model saved me a lot of time by generating initial drafts of the architecture, state diagrams, and flow charts, which I could then refine. It also helped me think about edge cases I might have missed on my own, such as security risks and recovery strategies. Having the rationale sections drafted with clear justifications gave me a stronger starting point to build a consistent and professional document.

That said, there were still parts that required significant human correction and reprompting. For example, the LLM sometimes produced overly complex diagrams or invented unnecessary classes, so I had to simplify them to keep the design realistic. I also needed to reframe prompts multiple times to make sure the outputs aligned with my original user story rather than drifting into unrelated details. In particular, the consistency across diagrams (architecture, class, state) was something I had to enforce manually, since the LLM sometimes generated mismatched labels or flows. Overall, the LLM was most useful for generating ideas quickly, but the polishing and alignment across sections relied heavily on human judgment.