Graphite: A Collaborative Canvas-Based IDE with CRDT Synchronization and Canvas-Aware Version Control

Hojun Lee (25-095) Jimin Kim (25-028)
Korea Science Academy of KAIST
2026-1 Information Science Project Showcase

Abstract. Graphite is a real-time collaborative canvas IDE that unifies visual architecture exploration with live code editing. Each file in a repository is represented as a draggable node on an infinite canvas; node positions, sizes, and file content are all synchronized across peers using Yjs Conflict-free Replicated Data Types (CRDTs). Graphite extends git with *canvas-aware version control*: each commit preserves the Yjs state snapshot, so branch checkout and diff visualization restore the exact spatial layout alongside file content. Additional contributions include an AI code-agent loop that applies edits as CRDT-safe incremental operations, full Language Server Protocol (LSP) integration for four languages, a dependency-graph heat map, and an annular bin-pack layout algorithm for collision-free node placement. Together these features allow distributed teams to collaborate on code structure and content simultaneously, bridging the gap between Figma-style visual collaboration and traditional IDE functionality.

1. Introduction

Traditional integrated development environments are fundamentally *linear*: code is accessed one file at a time through a flat tree list, making it difficult to grasp the overall structure and logic flow of a codebase at a glance. Tracking related components requires repeatedly navigating between files, creating significant cognitive overhead in dependency exploration. At the same time, distributed teams editing asynchronously cannot see where collaborators are working or what context surrounds their changes, which leads to large-scale merge conflicts and high collaboration cost during integration.

Graphite was built to address both of these problems through three core development goals:

1. **2D spatial work environment.** An infinite ReactFlow canvas with collapsible Region nodes spatially groups code hierarchies so that the entire codebase can be explored like a terrain map. Import relationships and parameter dependencies are rendered as edges, making structural knowledge visible rather than implicit.
2. **Real-time conflict-free collaboration.** A CRDT engine without central locking prevents concurrent editing conflicts at both the text and canvas levels. A *Spatial VCS* extends git to version-control not only file content but also the 2D placement of every node, enabling diff and merge operations on canvas layout alongside code.
3. **LLM-based code stability.** Previous real-time collab models failed due to instability of features such as autocomplete served by LSP while peer is typing thus producing intermediate— *broken* code. When concurrent edits break the Abstract Syntax Tree (AST), the watchdog detects the error and requests an LLM repair, restoring a valid parse tree without interrupting the active editing session.

Together these goals expand the development environment from a linear text editor into a collaborative 2D canvas IDE accessible to both developers and non-technical team members.

2. Main Body

2.1 Architecture

Graphite follows Electron's two-process model (Figure 1). The **main process** (`main.cjs`) runs in Node.js and owns all privileged operations: file I/O, directory scanning, spawning language servers, git operations, and AI agent execution. It exposes these capabilities to the renderer exclusively through typed IPC handlers.

The **preload script** (`preload.cjs`) bridges the process boundary by using Electron's `contextBridge` to inject `window.electronAPI` into the sandboxed renderer, providing a safe typed surface over IPC.

The **renderer process** (`src/`) is a React 18 single-page application. ReactFlow 11 provides the infinite canvas with drag-and-drop node management. Inside each node, a Monaco Editor instance renders file content. All mutable state — node positions, metadata, and file text — is held in a **Yjs** document rather than React state, so every mutation is automatically CRDT-replicated. The Yjs document is persisted per-room via `y-indexeddb` and synchronized peer-to-peer via `y-webrtc`.

The data flow for a node move is: `useYNodes` writes the new position to `Y.Map` → Yjs encodes a state update → `y-webrtc` broadcasts it to all peers → each peer's ReactFlow re-renders with the new position.

2.2 Software and Hardware Technologies

Table 1 lists the key technologies. Hardware requirements are modest: any x86-64 Linux or macOS desktop suffices; no GPU is required.

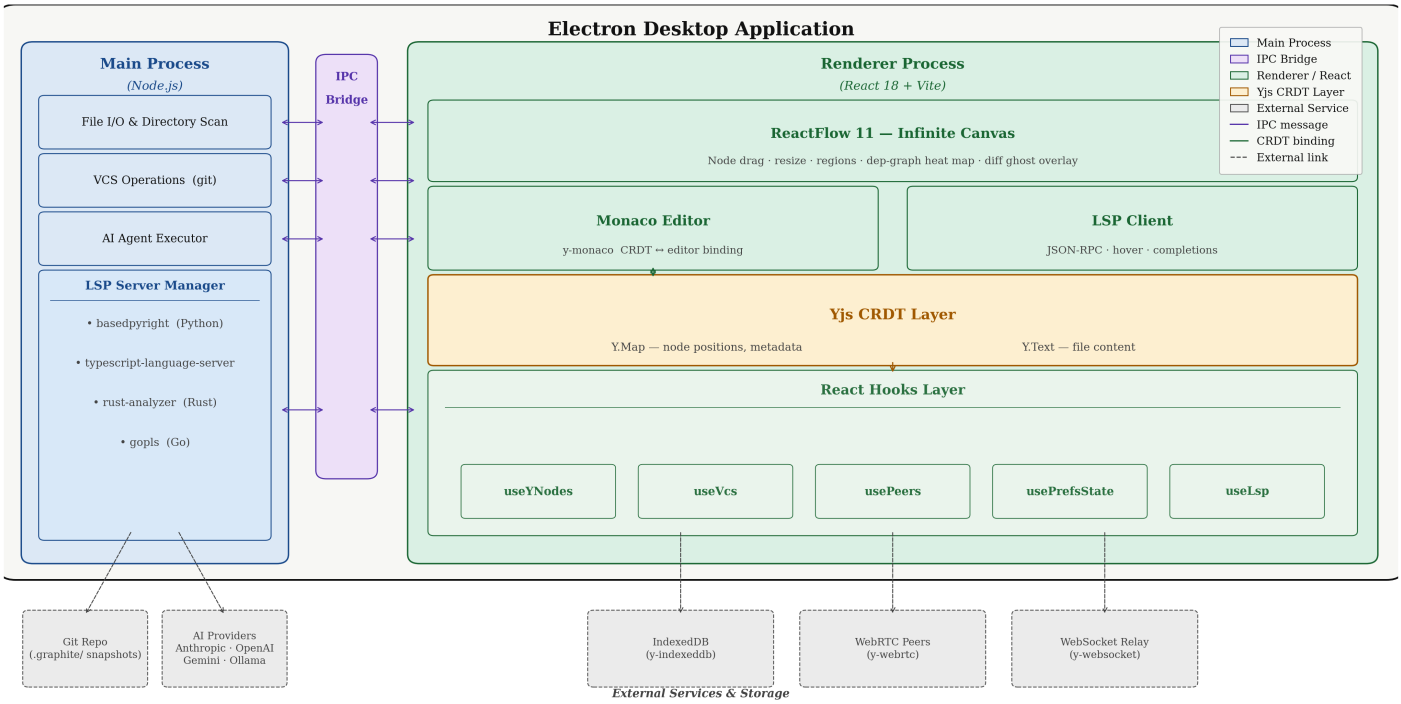

Figure 1: System architecture of Graphite. The main process exposes file, LSP, VCS, and agent IPC to the sandboxed renderer. The Yjs CRDT layer synchronizes canvas state across peers via WebRTC and persists it in IndexedDB.

Table 1: Software technologies used in Graphite.

Technology	Ver.	Role
Electron	33	Desktop shell, IPC, file I/O
React	18	UI component rendering
ReactFlow	11	Node-based infinite canvas
Monaco Editor	4.7	Embedded VS Code editor
Yjs	13.6	CRDT engine (`Y.Map`, `Y.Text`)
y-indexeddb	9.0	Per-room IndexedDB persistence
y-webrtc	10.3	P2P real-time CRDT sync
y-monaco	0.1	CRDT ↔ Monaco binding
basedpyright	latest	LSP: Python
ts-language-server	latest	LSP: TypeScript/JavaScript
rust-analyzer	latest	LSP: Rust
gopls	latest	LSP: Go
Anthropic SDK	0.98	AI agent (Claude tool-use loop)
diff-match-patch	1.0	CRDT-safe draft application
Git	—	VCS: branch, commit, merge

2.3 Implementation Methods and Algorithms

2.3.1 CRDT-Backed Canvas State

The core of Graphite's collaboration model is a singleton Yjs `Y.Doc` maintained in `src/crdt/doc.js`. Node positions and metadata are stored in a `Y.Map` (one entry per node ID); file content is stored in per-path `Y.Text` instances.

`useYNodes.js` replaces ReactFlow's standard `useNodesState` hook. Before any node write, `stripCallbacks()` removes React callback references and transient UI fields that must not enter the CRDT. The hook subscribes to `Y.Map` observation events and feeds the resulting plain objects back into ReactFlow as the node array.

Monaco editors are bound to `Y.Text` via `monacoBinding.js`, which calls the `y-monaco` `MonacoBinding` API and dynamically injects per-peer CSS rules to render colored named cursors. Each peer's cursor position is additionally broadcast through Yjs *Awareness* for the title-bar peer cluster.

Room codes are derived as `sha1(rootPath + ':' + branchName)`, making them stable across sessions and unique per branch.

2.3.2 Canvas-Aware Version Control

Standard git commits record only file text. Graphite augments each commit with a binary Yjs state blob stored in `refs/notes/graphite-snapshot`, written as a base64-encoded `Y.encodeStateAsUpdate()` call.

Branch checkout proceeds in four steps: (1) the main process runs `git checkout` and reads the snapshot note for the target branch; (2) a new Yjs room is initialized via `initRoom(rootPath, roomCode)`; (3) the snapshot blob is applied to the fresh doc with `Y.applyUpdate()`, restoring exact node positions; (4) the renderer animates a "lift-and-reveal" CSS transition (180 ms lift, 330 ms reveal) to signal the context switch.

Diff visualization uses `computeNodeDiff(baseNodes, currentNode` (`src/vcs/computeNodeDiff.js`), which returns `Map<id, {type, node, prevPosition}>` with types `added`, `removed`, `moved`, and `modified`. `FlowCanvas` injects semi-transparent `DiffGhostNode` overlays at the base positions; a `DiffPanel` shows per-type counts and an opacity slider.

3-way canvas merge (`src/vcs/mergeUtils.js`) applies standard 3-way merge semantics to node arrays: nodes changed on both sides become *conflicts* resolved via a per-node ours/theirs chooser; nodes added only on the source branch are auto-accepted; nodes deleted only on the source branch trigger a warning. When no common ancestor exists, the algorithm falls back to a 2-way merge. This is visualized by `computeNodeDiff.js`, the code for diff visualization mentioned above.

2.3.3 Annular Bin-Pack Layout Algorithm

When a directory is first opened, Graphite must place nodes without overlap. The algorithm in `src/utils/layout.js` works as follows:

1. For 2–4 items, use hardcoded presets (line, triangle, square) for aesthetic compactness.
2. For 5+ items, sort nodes by bounding-circle radius descending.
3. For each node to place, generate *orbit candidates*: points at distance ($r_{\text{placed}} + r_{\text{new}} + \text{padding}$) around each already-placed node.
4. Select the candidate closest to the origin that is free of collisions.
5. Slide the chosen position 2 px toward the origin repeatedly until a collision would occur (*tightening pass*).

A two-phase tree layout (`src/utils/treeLayout.js`) complements this: phase 1 estimates node sizes from file extension heuristics; phase 2 re-runs the algorithm after ReactFlow reports actual measured dimensions, correcting any overlaps introduced by estimation error.

2.3.4 AI Agent with CRDT-Safe Edit Application

The AI agent subsystem (`lib/agent/`) supports four providers: Anthropic (Claude, default `claude-sonnet-4-6`), OpenAI, Google Gemini, and Ollama for local models. The agent loop iterates until `stop_reason` $\neq$ `'tool_use'`. Available tools are: `read_file`, `edit_file`, `create_node`, `delete_node`, and `run_command`.

Edits use a *draft-then-accept* flow to avoid destructive writes. When the agent calls `edit_file`, the proposed content is stored as a *draft*; the `AgentPanel` renders a diff card for the user to accept or reject. On acceptance, `src/agent/applyDraft.js` computes a `diff-match-patch` diff between the current `Y.Text` content and the draft, then applies the diff as a sequence of Yjs `insert`/`delete` operations inside a single transaction. Because each operation is a minimal CRDT delta rather than a full replacement, concurrent peer edits are preserved correctly.

An **LSP Watchdog** (`src/lsp/watchdog.js`) complements the agent: it debounces LSP diagnostics by 1 s, calls `agentRepair()` if errors remain, and applies the repaired text as a *shadow document* to the LSP session without altering the visible editor content, keeping error highlighting accurate without interrupting the user.

3. Execution Results

Scenario 1: Real-Time Multi-User Collaboration

Two Electron windows are opened against the same workspace directory. Both derive the same room code from the path, so their Yjs documents automatically synchronize via `y-webrtc`. When User A drags a file node to a new position, User B's canvas updates within one WebRTC round-trip (<100 ms on a local network). When User A edits code inside a Monaco editor, User B sees each keystroke appear with a named, colored cursor rendered by the `y-monaco` binding and the per-peer CSS injection in `monacoBinding.js`.

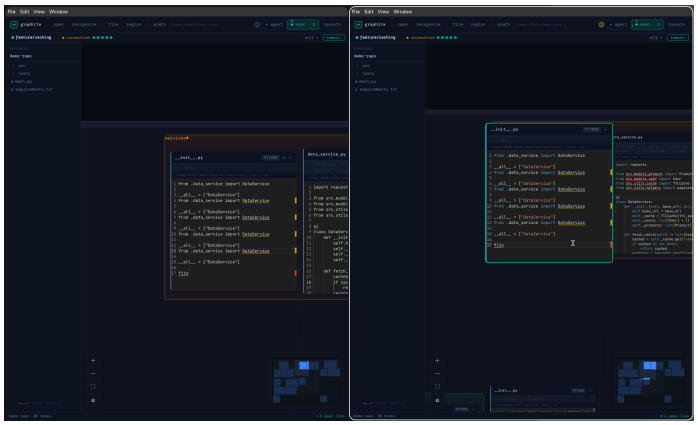

Figure 2: Real-time collaboration: two Electron windows sharing the same Yjs room. Node moves and keystrokes from User A appear on User B's canvas within a single WebRTC round-trip.

Scenario 2: Canvas VCS — Branch Diff and Merge

Starting from the `main` branch, the user creates a branch `feature/auth`, rearranges several file nodes into a new region group, adds two new nodes, and commits. Switching back to `main` and clicking "Diff vs. feature/auth" invokes `computeNodeDiff`: added nodes appear with a green tinted border; removed nodes appear as translucent red ghost overlays at their former positions; moved nodes show a faint arrow from the old position to the new one. Initiating a merge opens the conflict resolution panel, which lists each conflicting node with a side-by-side card showing "Ours" and "Theirs" positions; choosing one immediately updates the Yjs document.

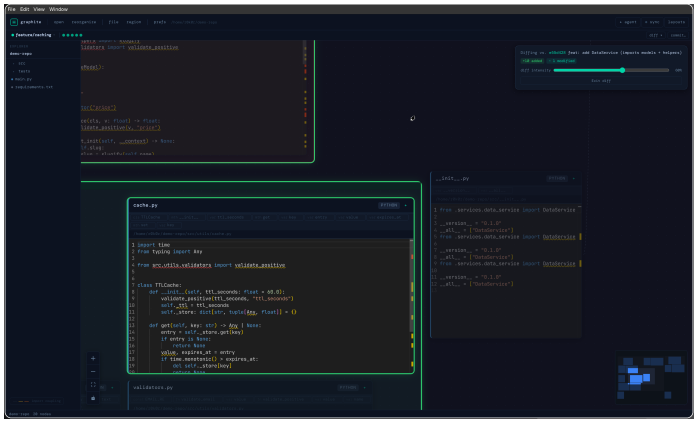

Figure 3: Canvas-aware diff and merge. Ghost nodes (red, semi-transparent) mark positions from the base snapshot. The merge panel (right) shows per-node conflict cards.

Scenario 3: AI Agent Editing with Dependency Graph

A Python project is opened; Graphite scans `import` statements and LSP document links to build a dependency graph. Edge thickness and color reflect reference frequency (the heat map): heavily imported modules have thick orange edges, rarely referenced ones have thin blue edges. The user opens the agent panel and types: *"Refactor `parse_args()` to use a dataclass instead of argparse."'* The agent calls `read_file` on the target module, produces a new version, and the draft card appears in the `AgentPanel`. After the user clicks Accept, `applyDraft` applies the diff as Yjs ops; Monaco re-renders the updated text; LSP re-indexes; and the dep-graph edges update live to reflect the new import structure.

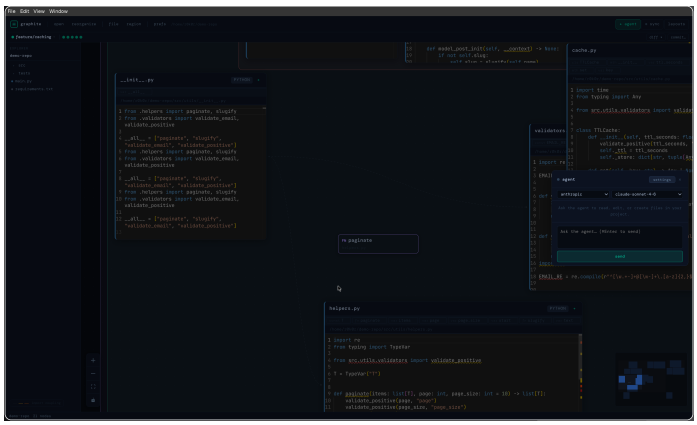

Figure 4: AI agent in action. The `AgentPanel` (left) shows the draft diff card; the canvas (right) displays the updated dependency-graph heat map after the edit is accepted.

4. Conclusion

Graphite demonstrates that a spatial, collaborative canvas can serve simultaneously as an architectural overview and a fully functional code editor. By making the Yjs CRDT the single source of truth for both node positions and file content, real-time multi-user collaboration is an inherent property of the system rather than a feature layered on top.

The canvas-aware VCS is, to our knowledge, a novel contribution: extending git to preserve spatial layout history enables a class of diff and merge operations that have no equivalent in existing tools. The annular bin-pack algorithm ensures that opening any repository produces a readable, collision-free initial layout without manual arrangement. The AI agent's use of `diff-match-patch` for CRDT-safe application of model-generated edits allows AI assistance to coexist safely with concurrent human edits.

Limitations. WebRTC connectivity is unreliable across strict NAT; a relay server (`y-websocket`) mitigates this but adds a centralized dependency. The LSP watchdog introduces up to one second of latency before repair triggers. Very large repositories (>500 files) stress the layout algorithm.

Future work includes a plugin API for custom node types, a GitHub Pull Request integration that renders canvas diffs in PR reviews, and a mobile read-only canvas viewer.

5. Others

5.1 Repository

https://github.com/ROKOR/graphite

5.2 Current Status and Future Work

All features described in this report are implemented and available on the `electron` branch. Planned extensions:

- GitHub PR visual diff integration
- Plugin API for third-party node types
- Mobile read-only canvas viewer
- Conflict-resolution history and rollback

5.3 Reflections

Building Graphite taught us that CRDT algorithms, while elegant in theory, demand careful attention at every system boundary: callbacks must be stripped before entering the shared document, and room isolation must be enforced per branch to prevent state bleed. The Electron process boundary proved to be both a strength and a constraint — it enforced a clean separation between UI and system access, but every new capability required explicit IPC through the preload script. We also underestimated the complexity of WebRTC in real deployments; reconnection, stale awareness entries, and NAT traversal required more defensive handling than initial prototypes suggested. The most rewarding outcome was seeing the dependency-graph heat map update live after an AI agent edit — a moment that validated the CRDT-first design and made the project feel genuinely novel.

Acknowledgments

We thank Korea Science Academy of KAIST for the opportunity to present this work at the 2026-1 Information Science Project Showcase.