ChatGPT

AI-Native Version Control System (ANVCS) – Architecture and Prototype Design

Overview: The proposed AI-native VCS layers AI-assisted features on top of a Git-like core, combining real-time CRDT collaboration, a directed-graph commit history, and rich agent context metadata. At runtime, multiple developers/agents can edit code concurrently via CRDTs, while commits record not only code snapshots but structured AI context (prompts, model versions, test results) as metadata. A semantic merge engine uses AST analysis or LLM reasoning to auto-resolve conflicts. The system remains operable locally with Git compatibility, yet is built to scale to cloud servers or peer networks for distributed agent teams.

Architecture Components

- **Real-Time CRDT Collaboration Engine:** A core component to synchronize live edits. Each file (or code buffer) is treated as a CRDT (e.g. an operation-based list CRDT like RGA) so that concurrent edits automatically converge without manual merges [1]. For example, using a framework like Collabs (a TypeScript CRDT library [2]) or similar, each agent's edit operations are broadcast (via WebSocket/WebRTC or a central server) and merged in causal order. This engine provides "Google-Docs style" live editing across agents [2] [1].

- **Commit DAG Manager:** Instead of a linear branch model, versions are maintained as a directed acyclic graph. Each commit node contains pointers to parent commits (one or more) and references the CRDT state (or diff) for the working files. The commit graph supports forks and merges naturally. Crucially, the graph structure itself can be treated as a CRDT: new commits (nodes/edges) propagate through the network and merge without conflict (as in Pijul's design, where adding vertices/edges yields a conflict-free datatype [3]). Each commit node includes the code snapshot plus **Agent Metadata** (see below). Like Git, commits are content-addressed (e.g. by hash of their contents and metadata) to ensure integrity and deduplication.

- **Agent Context Metadata Store:** Each commit is enriched with structured metadata: **prompt text**, **AI model/version**, **tooling versions**, and **test outcomes**. For instance, every commit object could embed a JSON record with fields such as `{"prompt": "...", "model": "gpt-4", "tests_passed": true, "coverage": 92, "agent_id": "...", "timestamp": "...", "notes": "Feature X added"}`. This realizes the "prompt+test bundle" paradigm advocated for AI-first workflows [4]. The metadata might be stored as Git commit message annotations or in sidecar files in the repo (e.g. a `.ai/` folder), but should use a standard schema so tools can query it (for example JSON or YAML with a defined schema). By default, every `ai-vcs commit` command auto-captures the active prompt and model version, along with CI test results (passing/failing). Recording these ensures the repository's history reflects not just *what* changed but *why* and *how* (which prompt or agent produced it [4]).

- **Semantic Merge Engine:** A service (or library) invoked whenever branches diverge or conflicts arise. Instead of naïve line diffs, this engine parses code into ASTs and merges based on syntax/semantics. For example, it can use tools like Tree-sitter or Roslyn to detect moved or renamed functions and reapply edits intelligently [5]. In complex cases, the engine can invoke an LLM:

given the two conflicting code versions and the original prompt context, it asks the model to reconcile them while preserving intent. Studies show AI can often resolve conflicts semantically by understanding developer intent [6]. The engine also runs automated tests on candidate merges to verify correctness before finalizing.

- **User Interface & CLI Layer:** Provides developer-facing tools. This includes a command-line interface (CLI) and plugins for IDEs/editors (e.g. a VSCode extension). The CLI offers commands reminiscent of Git but extended (see **"Commands"** below). IDE plugins hook into the CRDT engine to show real-time cursors and changes, and present diffs of intent rather than raw text lines. A visualization layer can display the full DAG of versions (beyond Git's linear branch view), aiding agent teams in tracking progress.

- **Storage & Sync:** Underneath, file contents and CRDT logs are stored locally (e.g. in the working directory and a `.ai/` metadata store) for offline-first use. For sharing across machines or the cloud, the system can use Git's object storage or a distributed database. One approach is to piggyback on Git packfiles for blobs and use Git's transfer protocols (HTTP/SSH) for the commit DAG. Alternatively, one can use a distributed log store or database (e.g. a CRDT-optimized NoSQL store like Apache CouchDB or a P2P store like IPFS) to replicate the commit graph and operations. The architecture allows either a centralized server (e.g. a cloud service hosting the master DAG and merge engine) or a fully peer-to-peer network (each agent publishes ops and pulls from peers, similar to Collabs or Pijul's gossip).

Diagram: Conceptually, the system layers are: 1. **Editor/Agent Clients** (with CRDT syncing and AI tools) 2. **Local VCS Layer** (commit graph + metadata) 3. **Network/Server** (synchronization, merge services) 4. **Storage** (Git object store or database).

Each layer interacts: clients send CRDT ops and commit commands → local engine updates the DAG and context store → remote server (or peers) merges DAGs and runs semantic merges → updates propagated back to clients.

Language, Data Stores, Protocols, and Standards

- **Languages:** Use languages popular in DevOps and AI. A compiled language like **Go** or **Rust** is ideal for the core VCS engine and CLI (for performance, static binaries, and safety). (For instance, Evo is implemented in Go [7] [8] and Pijul in Rust, showing that such systems benefit from strong typing and concurrency support.) A high-level language like **TypeScript/JavaScript** is suitable for the CRDT collaboration layer (Collabs is TS [9]) and editor plugins, given the wealth of web/IDE tooling. **Python** can be used for the AI orchestration (LLM prompts, metadata processing, invoking tests) due to its rich ML libraries. Overall, a polyglot approach is fine: e.g. Go for core, TS for CRDT front-end, Python/Node for integration scripts.

- **Data Stores:** - *Content store:* Continue to use Git's content-addressable object store (packfiles) for file blobs, ensuring compatibility. - *Commit DAG and metadata:* A lightweight graph store or embedded database (e.g. SQLite, RocksDB) can hold commit nodes and metadata. This supports indexing/search of prompts or test results. For scaling, a distributed datastore (e.g. DynamoDB, CockroachDB, or a blockchain/DAG store) could replicate the commit graph globally. - *CRDT log storage:* Each file's CRDT operation log could be stored as an append-only file (or a table in a database). Alternatively, leverage existing CRDT backends (Automerge's JSON format or Yjs) which serialize the state. - *Tests and artifacts:* Use standard formats (e.g. JUnit XML, JSON) in a CI artifact storage (e.g. GitHub Actions artifacts or an S3 bucket) referenced by commit metadata.

- **Protocols:** - *Synchronization:* Use Git's own push/pull (SSH, HTTPS) for general sharing of the commit DAG and blobs, so users can still `git push` to a remote. For real-time CRDT syncing, use WebSockets or WebRTC (as Collabs does) or gRPC streams. - *APIs:* Expose a REST or gRPC API for actions like "submit CRDT ops" or "request merge"; this could integrate with CI/CD pipelines. - *Authentication:* Standardize on SSH keys or OAuth (GitHub/GitLab auth) for agent identity. - *Open Formats:* Store all metadata in JSON or YAML (common standards). Use a schema or JSON-LD context so fields like "prompt" or "model" are well-defined. For ASTs, use the standard parse trees (e.g. an ESTree or Tree-sitter JSON).

- **Open Standards:** - *Git:* Leveraging Git itself (its on-disk format and network protocols) ensures broad adoption. - *CRDT Libraries:* Build on established CRDT libraries (Automerge, Yjs, or Collabs) which follow W3C/industry conventions for conflict-free types. - *AST formats:* Rely on open language grammars (Tree-sitter, ANTLR) for syntax trees. - *Provenance:* Optionally use W3C PROV or similar for recording agent actions. - *Model Context:* If available, use standards like OpenAI's fine-tuning or model cards to describe model versions. - *Security:* Use OpenPGP or Ed25519 signatures on commits (as Evo suggests) to authenticate agent contributions [7].

Key Data Structures

- **Change Model (CRDT ops):** Each file is backed by a CRDT (e.g. an *operation-based list CRDT*). For text files, operations are "insert line N with content X" or "delete line N". These ops carry causal metadata (e.g. vector clock or lamport timestamp). The system can also support higher-level ops (e.g. "rename function foo→bar") captured by AST diff. All ops are commutative and mergeable [2]. In practice, we maintain an operation log per file: a sequence of ops with IDs. When syncing, replicas merge their logs (union of ops). The current file state is the result of applying all ops in topological order.

- **Commit DAG Representation:** Commits are nodes in a directed acyclic graph. Each commit record contains:

- A unique ID (hash of its contents and metadata).
- Parent ID(s) (one for a normal commit, multiple for a merge).
- A pointer to the CRDT state or diff (for example, a snapshot hash or a reference to the last operation index for each file).
- The **Agent Metadata** payload (see below).

- A human message (author message and AI prompt). The DAG is append-only: new commits attach new nodes and edges. No commit is ever deleted. This matches traditional VCS, but unlike Git's linear "branch pointer" model, branches are just named references to certain commits in the DAG. (Evo calls them "streams" to emphasize flexibility [7].) We also store a mapping of branch/stream names to commit IDs. The entire DAG can be stored in a graph database or in files (as Pijul's model illustrates, the commit graph itself can act as a CRDT [3], so it can merge across replicas).

- **Agent Metadata Format:** A JSON/YAML schema such as:

```
commit_id: abc123
prompt: |
  Write a function to sort a list in Python
model: gpt-4-turbo
```

```
agent_type: LLM     # or 'human', or agent name
timestamp: 2025-11-01T12:00:00Z
tests_passed: true
test_summary:
  total: 42
  passed: 42
coverage: 85.2
commit_summary: "Added sort_list function with AI assistance"
```

This metadata is automatically attached to each commit. Using YAML (or JSON) ensures it's machine-parseable, and by citing this explicitly as part of the commit, we follow the vision that "versioning the prompt and tests" becomes the core unit of history [4]. Search tools can index these fields (e.g. find all commits by a certain model or see which prompt produced failing tests).

Interfaces and CLI Commands

The system provides both Git-style CLI commands and richer AI-specific commands. Example workflows:

- **Initialization:**

```
ai-vcs init
```

Initializes a new repository (or upgrade an existing `.git` repo). This creates a `.ai/` folder for CRDT logs and metadata, and configures Git hooks.

- **Adding and Committing:**

```
ai-vcs add file1.py file2.py
ai-vcs commit -m "Implement feature X" \
    --prompt "Refactor this code for efficiency using AI" \
    --model "gpt-4-turbo" \
    --test-report tests/results1.xml
```

This records changes to files (`add` works like `git add`) and creates a commit. The CLI automatically captures the given prompt and model, runs the specified test suite, and includes the test results and coverage in the commit metadata. It then writes a commit to the DAG. Under the hood this could call `git commit` with a special formatted commit message or use Git notes to store the AI metadata.

- **Branch/Stream Management:**

```
ai-vcs branch create experiment-foo
ai-vcs switch experiment-foo
```

(Or use `stream` as Evo does.) Branches are lightweight names pointing to DAG nodes. Creating a branch does not copy content, it just names the current commit.

- **CRDT Collaboration:**
  Agents can join a real-time session:

```
ai-vcs collab start
# start a collab session (opens a websocket server or room)
ai-vcs collab join code_repo://room123  # join others
```

  All edits in this mode bypass the normal `add/commit` cycle: every keystroke generates a CRDT operation that is broadcast and merged immediately. This emulates a live editor. Optionally, `ai-vcs collab commit` can snapshot the current state into a commit node when desired.

- **Merging:**

```
ai-vcs merge feature-branch into main
```

  This attempts a merge of two heads. If changes are disjoint, it auto-merges; if there are conflicts, the AI merge engine intervenes. The tool can offer choices: for each conflict, present both versions and the AI's suggested resolution, along with test outcomes. Once accepted, it creates a new merge commit (with two parents) in the DAG. Semantic merge (AST-level) is the default strategy [5]; an `--auto` flag can let the AI pick without user intervention, or a `--test-check` flag can enforce that the merge passes all tests.

- **Syncing and Remotes:**

```
ai-vcs push origin main
ai-vcs pull origin feature-branch
```

  These use standard Git remotes. The system can map its commit DAG onto Git refs so that pushing works with a normal Git server. For example, each DAG commit could also produce a real Git commit in a hidden branch. This ensures that existing infrastructure (GitHub, CI) still works. The `.ai/` metadata might be pushed as separate ref or included in commit messages/notes.

- **Browsing History:**
  Commands like `ai-vcs log` or `ai-vcs graph` show the DAG, including branch labels and metadata (e.g. annotate each commit with its prompt snippet and model). Because the graph can be very large, the UI should allow filtering (e.g. show only commits by a particular agent or only merge commits).

Each command hides the complexity: to the user, it feels like Git but "smarter". For example, when creating a PR or pushing, the system can auto-generate release notes by summarizing commit prompts and changes via an LLM (since the prompts describe intent). The goal is minimal disruption of developer workflows: familiar command names plus new AI flags.

Integration with Existing Git Workflows

To encourage adoption, this design **layers on top of Git** rather than replacing it. Key strategies:

- **Git Compatibility Layer:** The simplest mode is as a Git extension. We implement all tools as `git-aisync`, `git-commit-ai`, etc., so running `git` still works. The AI-specific commands are invoked through `git aisync ...` but internally operate on the same repo. Even in AI mode, any commit should also create a valid Git commit (possibly with the metadata in the message or notes) so that standard Git can clone/fetch the repository. Thus one can always fallback to plain Git if needed.

- **Augmented Commit Objects:** Use Git commit objects to store the code snapshot and author info, and store the AI context either in the commit message (in a structured YAML header) or via `git notes`. For example, the commit message might begin with:

  ```
  ai-meta: {prompt: "...", model: "...", tests_passed: true}
  ```

  followed by the normal description. This way, the AI metadata travels with the commit.

- **Branch Semantics:** The system can use Git branches to represent "streams" or workspaces, but it also allows more fluid branching (e.g. ephemeral branches for ideas that automatically merge). By default, developers can still use `git branch` and `git merge`, but the `ai-vcs` commands will advise or automate those operations.

- **Editor & CI Integration:** The IDE plugins for VSCode/Git can intercept saves and trigger `ai-vcs` commits. CI/CD pipelines (e.g. GitHub Actions) can recognize AI-metadata: for instance, an action can display the stored prompt for each commit, or automatically run semantic checks on merges. The system can also publish a GitHub web API that shows the AI log.

- **Migration Path:** Existing Git repos can be gradually migrated: one could install the AI extension without rewriting history. On first use, the tool scans the Git log and attaches empty context to old commits, then new commits start capturing context. This lowers the barrier to try AI features.

Recommendations for Adoption and Scalability

- **Language Choice:** Use **Go or Rust** for the core engine (fast, safe binaries); use **TypeScript** for any UI/CRDT libraries (Collabs is TS [9]); use **Python** for LLM orchestration. This mixes stable compiled code (suitable for CLI and server) with flexible scripts for AI calls.

- **Data Stores:** Store commit DAG and metadata in a robust local database (SQLite or embedded RocksDB). For team/cloud use, replicate to a scalable DB (Cassandra, CockroachDB) or to Git hosting. CRDT logs can just be JSON/Protobuf files appended per edit. Avoid proprietary formats; stick to open standards (Git's pack, JSON, YAML, IPFS content hashes).

- **Protocols & APIs:** - Use Git's wire protocol (SSH/HTTPS) for pushing/pulling history. - Use standard REST/gRPC APIs for any server components (e.g. a merge service or context index). -

For editor collaboration, use WebSocket-based CRDT sync (as Collabs and others do). - Support existing Git workflows (e.g. HTTP GitHub endpoints) so that normal Git clients can still operate.

- **Open Standards:** - **Diff/AST:** Base merges on established AST formats (e.g. Babel AST for JS, Spoon for Java). - **CRDT:** Leverage existing CRDT specs (e.g. the Automerge or Yjs data models) so that future tools can interoperate. - **Metadata:** Define a clear schema (open repo's JSON schema) for AI context, so other tools (CI, dashboards) can read it. - **Authentication:** Use SSH keys or OAuth (as Git does) so no new login is needed.

Example Data Flows

1. **Local Edit & Commit:** Alice (an AI agent or dev) edits code in her IDE. The CRDT engine logs her keystroke ops. She invokes `ai-vcs commit`, supplying her prompt. The system merges her local CRDT ops into the current branch state, runs tests, and creates a commit node containing the diff+metadata. This also calls `git commit` under the hood, so a Git SHA is produced.

2. **Remote Sync & Merge:** Bob fetches from the remote. His client pulls new CRDT ops and new commits into his local store. Since Alice used CRDTs, Bob sees her edits already merged into his local files. When he runs `ai-vcs merge feature1 into main`, the semantic merge engine runs (parsing ASTs). It finds no syntactic conflicts, so it creates a merge commit in the DAG. The combined commit (with both parents) is then pushed.

3. **Agent Collaboration:** An automated AI agent (CI bot) notices a new commit with failing tests. It automatically opens `ai-vcs collab join roomXYZ` where other bots are reviewing. It uses the semantic merge tool to propose a fix, adds a CRDT edit, and then runs `ai-vcs commit --auto-fix`. This commit includes the prompt "Fix failing tests" and model info. The test suite now passes.

Integration Summary

By building on Git's model (content-addressing, DAG history, and network protocols) while adding new layers for AI and CRDTs, this system maximizes familiarity. Existing `git` commands still work, but AI-aware subcommands automate many tasks. Semantic merging and context capture make merges less error-prone and version history more informative, as envisioned in recent developer trends [10] [4]. In essence, the system acts as a drop-in upgrade to Git: users keep their tools (editors, CI, remotes) but gain an "AI copiloting" layer that tracks and reasons about their work.

References

This design is informed by recent research and projects in CRDT-based version control and AI-assisted workflows. For example, Collabs [2] and Evo [7] demonstrate how CRDT frameworks can provide conflict-free text merging and named "streams" instead of rigid branches. Pijul's theory shows a DAG-based change history can be made conflict-free by modeling changes as graph edits [3]. Upwelling illustrates blending real-time edits with version control drafts [11]. Industry insights highlight the need to store prompts and test results as part of history [4] and to leverage semantic AST merges for intelligent conflict resolution [5] [6]. These and other ideas were incorporated to ensure the architecture is both practical and forward-looking.

---

1 7 8 GitHub - binbandit/evo: Version control that works the way you think
https://github.com/binbandit/evo

2 9 arxiv.org
https://arxiv.org/pdf/2212.02618

3 Pijul - Towards 1.0
https://pijul.org/posts/2020-11-07-towards-1.0/

4 Emerging Developer Patterns for the AI Era | Andreessen Horowitz
https://a16z.com/nine-emerging-developer-patterns-for-the-ai-era/

5 Merging Done Right: Semantic Merge - DaedTech
https://daedtech.com/merging-done-right-semantic-merge/

6 10 AI Native Version Control - Let AI Tools Manage Your Git Workflow
https://zenvanriel.nl/ai-engineer-blog/ai-native-git-workflow-automation/

11 Upwelling: Combining real-time collaboration with version control for writers.
https://www.inkandswitch.com/upwelling/