Solo Git Architecture

Comprehensive System Architecture Documentation

Table of Contents

System Overview

Solo Git is built as a layered architecture with clear separation of concerns:

```
┌─────────────────────────────────────────────────────────────┐
│                    User Interfaces                           │
│  ┌──────────┐    ┌──────────┐    ┌──────────────────┐       │
│  │   CLI    │    │   TUI    │    │   Desktop GUI    │       │
│  │ (Rich)   │    │(Textual) │    │  (Tauri+React)   │       │
│  └────┬─────┘    └────┬─────┘    └────────┬─────────┘       │
└───────┼───────────────┼───────────────────┼─────────────────┘
        │               │                   │
        └───────────────┼───────────────────┘
                        │
┌───────────────────────▼─────────────────────────────────────┐
│                 State Manager                                │
│  ┌──────────────────────────────────────────────────┐      │
│  │  JSON-based state (repos, workpads, tests, AI)   │      │
│  │  Bidirectional sync with Git and filesystem      │      │
│  └──────────────────────────────────────────────────┘      │
└─────────────────────────┬───────────────────────────────────┘
                          │
┌─────────────────────────▼───────────────────────────────────┐
│                    Core Engine Layer                         │
│  ┌─────────┐  ┌────────────┐  ┌───────────────┐           │
│  │  Git    │  │   Patch    │  │     Test      │           │
│  │ Engine  │  │  Engine    │  │ Orchestrator  │           │
│  └─────────┘  └────────────┘  └───────────────┘           │
└─────────────────────────┬───────────────────────────────────┘
                          │
┌─────────────────────────▼───────────────────────────────────┐
│              AI Orchestration Layer                          │
│  ┌──────────────────────────────────────────────────┐      │
│  │  Model Router  │  Planning Engine  │ Code Gen   │      │
│  │  Cost Guard    │  Test Analyzer    │            │      │
│  └──────────────────────────────────────────────────┘      │
└─────────────────────────┬───────────────────────────────────┘
                          │
┌─────────────────────────▼───────────────────────────────────┐
│                Workflow Automation                           │
│  ┌─────────────┐  ┌──────────────┐  ┌────────────┐        │
│  │ Auto-Merge  │  │ CI/CD        │  │  Rollback  │        │
│  │   Gate      │  │ Orchestrator │  │  Handler   │        │
│  └─────────────┘  └──────────────┘  └────────────┘        │
└─────────────────────────┬───────────────────────────────────┘
                          │
┌─────────────────────────▼───────────────────────────────────┐
│              External Services                               │
│  ┌──────────────┐     ┌────────────┐                       │
│  │  Abacus.ai   │     │  Jenkins   │                       │
│  │  RouteLLM    │     │  (Optional)│                       │
│  │   API        │     └────────────┘                       │
│  └──────────────┘                                           │
└─────────────────────────────────────────────────────────────┘
```

Architecture Principles

1. **Separation of Concerns**: Clear boundaries between UI, business logic, and data
2. **Event-Driven**: Components communicate via events and state changes
3. **API-First**: All operations exposed via internal APIs
4. **Idempotent Operations**: Safe to retry, no unexpected side effects
5. **Fail-Safe**: Errors leave system in consistent state
6. **Observable**: Comprehensive logging and metrics

Core Components

1. Repository Core (`sologit/core/`)

Purpose: Fundamental data models and business logic

Repository (`repository.py`)

- Represents a Git repository with Solo Git enhancements
- Tracks workpads, trunk state, and configuration
- Manages repository lifecycle (init, clone, archive)

```
class Repository:
    def __init__(self, path: Path, config: Config):
        self.path = path
        self.trunk = "main"  # Protected branch
        self.workpads: Dict[str, Workpad] = {}
        self.state = StateManager(path)

    def create_workpad(self, title: str) -> Workpad:
        """Create ephemeral workpad from trunk"""

    def promote_workpad(self, workpad_id: str) -> CommitResult:
        """Fast-forward merge workpad to trunk"""
```

Workpad (`workpad.py`)

- Ephemeral workspace (replaces branches)
- Auto-named, disposable, isolated from trunk
- Tracks patches, test results, and AI interactions

```
class Workpad:
    def __init__(self, id: str, base_commit: str):
        self.id = id
        self.title = title
        self.base_commit = base_commit  # Trunk SHA
        self.patches: List[Patch] = []
        self.test_results: List[TestResult] = []
        self.status: WorkpadStatus = WorkpadStatus.ACTIVE

    def apply_patch(self, patch: Patch) -> ApplyResult:
        """Apply code changes to workpad"""

    def can_promote(self) -> bool:
        """Check if workpad is ready for trunk merge"""
```

2. Engine Layer (`sologit/engines/`)

Git Engine (`git_engine.py`)

- Low-level Git operations wrapper
- Fast-forward merge enforcement
- Conflict detection and resolution
- Repository integrity checks

Key Operations:

```python
class GitEngine:
    def fast_forward_merge(self, target: str, source: str) -> MergeResult:
        """Fast-forward only merge"""

    def create_ephemeral_branch(self, name: str, base: str) -> Branch:
        """Create workpad branch"""

    def rebase_on_trunk(self, branch: str) -> RebaseResult:
        """Update workpad with latest trunk changes"""

    def detect_conflicts(self, branch1: str, branch2: str) -> List[Conflict]:
        """Check for merge conflicts"""
```

Patch Engine (`patch_engine.py`)

- Unified diff generation and application
- Conflict detection and resolution
- Patch validation and testing
- Rollback capabilities

```python
class PatchEngine:
    def generate_patch(self, code_changes: CodeChanges) -> Patch:
        """Generate unified diff from AI code changes"""

    def apply_patch(self, patch: Patch, target_path: Path) -> ApplyResult:
        """Apply patch with conflict detection"""

    def validate_patch(self, patch: Patch) -> ValidationResult:
        """Check patch syntax and applicability"""
```

Test Orchestrator (`test_orchestrator.py`)

- Test execution in isolated sandboxes
- Parallel test running
- Result aggregation and reporting
- Failure diagnosis

```python
class TestOrchestrator:
    def run_tests(
        self,
        target: TestTarget,
        workpad: Workpad
    ) -> TestResult:
        """Execute tests in sandbox"""

    def run_parallel(self, tests: List[Test]) -> List[TestResult]:
        """Run multiple tests concurrently"""

    def analyze_failures(
        self,
        results: List[TestResult]
    ) -> FailureAnalysis:
        """Diagnose test failures with AI"""
```

3. AI Orchestration (`sologit/orchestration/`)

Model Router (`model_router.py`)

- Intelligent model selection based on task type
- Multi-model support via Abacus.ai RouteLLM
- Automatic fallback on model failures
- Model performance tracking

Model Selection Logic:

```python
class ModelRouter:
    def select_model(
        self,
        task: Task,
        context: Context
    ) -> Model:
        """Select optimal model for task"""

        # Escalation rules
        if task.is_security_sensitive():
            return self.planning_model  # GPT-4/Claude
        if task.complexity_score() > 0.7:
            return self.planning_model
        if task.patch_size() > 200:
            return self.planning_model
        if task.test_failures >= 2:
            return self.planning_model

        # Normal routing
        if task.type == TaskType.PLANNING:
            return self.planning_model
        elif task.type == TaskType.CODING:
            return self.coding_model
        else:
            return self.fast_model
```

Planning Engine (`planning_engine.py`)

- High-level task decomposition
- Architecture and design decisions
- Complex problem solving
- Strategic code changes

Code Generator (`code_generator.py`)

- Patch generation from prompts
- Refactoring and optimization
- Boilerplate generation
- Documentation writing

Cost Guard (`cost_guard.py`)

- API usage tracking by model
- Daily budget enforcement
- Cost alerts and notifications
- Usage analytics

```python
class CostGuard:
    def track_request(
        self,
        model: str,
        tokens: int
    ) -> CostResult:
        """Track API usage and cost"""

    def check_budget(self, estimated_cost: float) -> bool:
        """Check if request is within budget"""

    def alert_threshold(self) -> None:
        """Send alert when approaching budget limit"""
```

Heaven Interface Architecture

Design Philosophy

Heaven Interface follows **Jony Ive** and **Dieter Rams** minimalist principles:
- **“As little design as possible”**
- Code is always center stage
- Tools are unobtrusive, revealed on demand
- Consistent, calm, focused experience

Three Modes

1. Enhanced CLI (Python Rich)

Components:
- `formatter.py` : Rich console formatting
- `graph.py` : ASCII commit graph renderer
- `theme.py` : Color scheme and styling

Features:
- Colored output with syntax highlighting
- Progress bars and spinners
- Tables and panels for structured data
- Live updates during long operations

```python
from rich.console import Console
from rich.table import Table

console = Console()

# Styled output
console.print("[bold green]✓[/] Tests passed", style="success")

# Commit graph
console.print("""
  ○ a1b2c3d (HEAD -> main) Add caching
  │
  ○ e4f5g6h Refactor search
  │
  ○ i7j8k9l Initial commit
""")
```

2. Interactive TUI (Textual)

Components:

- `heaven_tui.py`: Main TUI application
- `tui_app.py`: Widget library
- `command_palette.py`: Fuzzy command search
- `file_tree.py`: File browser widget
- `test_runner.py`: Live test results
- `autocomplete.py`: Command completion
- `history.py`: Command history

Layout:

```
┌──────────────────────────────────────────────────────────┐
│  Solo Git - Heaven TUI                          [?] Help │
├──────────────────────────────────────────────────────────┤
│  [Ctrl+P] Command Palette  [Ctrl+K] Shortcuts            │
├──────────────┬───────────────────────────┬───────────────┤
│ File Tree    │  Code / Output Area       │  Commit       │
│              │                           │  Graph        │
│ ▼ src/       │ # main.py                 │               │
│   ├─ cli/    │ def run():                │   ● main      │
│   ├─ core/   │     ...                   │   │           │
│   └─ ui/     │                           │   ● feat      │
│              │                           │   │           │
│ ▶ tests/     │                           │   ● fix       │
│              │                           │               │
├──────────────┴───────────────────────────┴───────────────┤
│  [Log] Tests running... ⠋                                │
└──────────────────────────────────────────────────────────┘
```

Key Features:

- Full keyboard navigation (vim-style bindings)
- Live updates (tests, git operations)
- Fuzzy search command palette
- Split panes (resizable)
- Theme customization

3. Desktop GUI (Tauri + React + TypeScript)

Technology Stack:
- **Frontend**: React 18 + TypeScript + Vite
- **Backend**: Rust (Tauri)
- **Editor**: Monaco Editor
- **Graphics**: D3.js (commit graph)
- **Charts**: Recharts (metrics)
- **State**: React Context + IPC bridge

Component Structure:

```
heaven-gui/
├── src/                      # React frontend
│   ├── App.tsx              # Main app component
│   ├── components/
│   │   ├── CodeEditor.tsx   # Monaco wrapper
│   │   ├── CommitGraph.tsx  # D3.js visualization
│   │   ├── FileTree.tsx     # File browser
│   │   ├── TestDashboard.tsx # Test results
│   │   ├── AIAssistant.tsx  # Chat interface
│   │   ├── CommandPalette.tsx
│   │   ├── StatusBar.tsx
│   │   └── SettingsPanel.tsx
│   ├── hooks/
│   │   ├── useStateSync.ts  # State synchronization
│   │   ├── useCommands.ts   # Command execution
│   │   └── useWebSocket.ts  # Live updates
│   ├── services/
│   │   ├── api.ts           # Backend API client
│   │   ├── state.ts         # State management
│   │   └── ipc.ts           # Tauri IPC bridge
│   └── styles/
│       ├── theme.css        # Heaven design tokens
│       └── components.css
│
└── src-tauri/               # Rust backend
    ├── src/
    │   ├── main.rs          # Tauri setup
    │   ├── commands.rs      # IPC commands
    │   ├── state.rs         # State bridge to Python
    │   └── events.rs        # Event emitter
    └── Cargo.toml
```

5 Engagement Levels:

1. **Level 0 - Idle/Preview**
   - Only code editor visible (full screen)
   - Minimal status bar
   - Like “theater mode” for code

2. **Level 1 - Navigation**
   - File tree or command palette overlays
   - Quick file switching
   - Code remains visible (dimmed)

3. **Level 2 - Planning**
   - AI assistant pane slides in (right rail)

   - User discusses changes with AI
   - Code editor still center stage

4. **Level 3 - Coding/Test**
   - Bottom rail shows live test results
   - Editor highlighted
   - Status indicators for tests

5. **Level 4 - Commit/Resolve**
   - All panels may be visible
   - Diff view for conflicts
   - Commit message input
   - CI status display

Design Tokens (from `styles/theme.css`):

```css
:root {
  /* Colors */
  --color-bg: #1E1E1E;
  --color-text: #DDDDDD;
  --color-accent-blue: #61AFEF;
  --color-accent-green: #98C379;
  --color-accent-red: #E06C75;

  /* Typography */
  --font-code: 'JetBrains Mono', 'SF Mono', monospace;
  --font-ui: 'SF Pro', 'Roboto', sans-serif;
  --font-size-code: 14px;
  --font-size-ui: 12px;

  /* Spacing (8px grid) */
  --space-xs: 8px;
  --space-sm: 16px;
  --space-md: 24px;
  --space-lg: 32px;

  /* Animation */
  --transition-fast: 150ms ease-in-out;
  --transition-normal: 300ms ease-in-out;
}
```

State Management

State Schema

Core State Structure (`sologit/state/schema.py`):

```python
@dataclass
class SoloGitState:
    """Root state object"""
    version: str
    repositories: Dict[str, RepositoryState]
    global_config: GlobalConfig
    ai_metrics: AIMetrics

@dataclass
class RepositoryState:
    id: str
    path: Path
    trunk: str
    workpads: Dict[str, WorkpadState]
    recent_commits: List[CommitInfo]
    test_history: List[TestResult]

@dataclass
class WorkpadState:
    id: str
    title: str
    status: WorkpadStatus  # ACTIVE, TESTING, READY, MERGED
    base_commit: str
    patches: List[PatchInfo]
    test_results: List[TestResult]
    ai_interactions: List[AIInteraction]
    created_at: datetime
    updated_at: datetime
```

State Synchronization

Bidirectional Sync Flow:

```
  Git Repository          State JSON         UI Components
       │                      │                     │
       │   commit detected    │                     │
       ├─────────────────────>│                     │
       │                      │   state change      │
       │                      ├────────────────────>│
       │                      │                     │
       │                      │   user action       │
       │                      │<────────────────────┤
       │   git operation      │                     │
       │<─────────────────────┤                     │
       │                      │                     │
       │   result             │   update state      │
       ├─────────────────────>├────────────────────>│
```

State Manager (`sologit/state/manager.py`):

```
class StateManager:
    def __init__(self, repo_path: Path):
        self.repo_path = repo_path
        self.state_file = repo_path / ".sologit" / "state.json"
        self.lock = threading.Lock()

    def load_state(self) -> SoloGitState:
        """Load state from JSON file"""

    def save_state(self, state: SoloGitState) -> None:
        """Persist state to JSON file"""

    def watch_for_changes(self, callback: Callable) -> None:
        """Watch state file for external changes"""

    def sync_with_git(self) -> None:
        """Sync state with actual Git repository"""
```

State File Location:

```
my-project/
├── .git/                    # Git metadata
├── .sologit/                # Solo Git state
│   ├── state.json           # Main state file
│   ├── config.yaml          # Repo-specific config
│   └── cache/               # Temporary data
└── [project files]
```

AI Orchestration

Multi-Model Architecture

Three-Tier System:

```
┌─────────────────────────────────────────────────────┐
│               Task Classification                    │
│  (Complexity Analysis, Security Check, Context)     │
└──────────────┬──────────────────────────────────────┘
               │
        ┌──────┴──────┬──────────┐
        │             │          │
        ▼             ▼          ▼
┌──────────┐   ┌──────────┐   ┌──────────┐
│ Planning │   │  Coding  │   │   Fast   │
│  Models  │   │  Models  │   │  Models  │
├──────────┤   ├──────────┤   ├──────────┤
│ GPT-4    │   │ DeepSeek │   │ Llama 3  │
│ Claude   │   │ CodeLlama│   │ Gemma 2  │
└──────────┘   └──────────┘   └──────────┘
     │              │               │
     └──────────────┼───────────────┘
                    │
          ┌─────────▼──────────┐
          │   Abacus.ai        │
          │   RouteLLM API     │
          │   (Single Endpoint)│
          └────────────────────┘
```

Task Classification Algorithm

```python
def classify_task(task: Task) -> ModelTier:
    """Determine which model tier to use"""

    # Security-sensitive → Planning model
    if has_security_keywords(task.prompt):
        return ModelTier.PLANNING

    # Large changes → Planning model
    if task.estimated_lines > 200:
        return ModelTier.PLANNING

    # Multiple test failures → Planning model
    if task.test_failures >= 2:
        return ModelTier.PLANNING

    # High complexity → Planning model
    complexity = calculate_complexity(task)
    if complexity > 0.7:
        return ModelTier.PLANNING

    # Explicit architecture/design prompts
    if is_architectural(task.prompt):
        return ModelTier.PLANNING

    # Standard coding tasks
    if task.type in [TaskType.REFACTOR, TaskType.FEATURE]:
        return ModelTier.CODING

    # Simple edits, docs, boilerplate
    return ModelTier.FAST
```

Cost Tracking

Per-Request Tracking:

```
@dataclass
class APIRequest:
    model: str
    prompt_tokens: int
    completion_tokens: int
    cost_usd: float
    latency_ms: int
    timestamp: datetime
    task_type: str
    success: bool
```

Daily Budget Enforcement:

```
class CostGuard:
    def check_budget(self, estimated_cost: float) -> BudgetResult:
        today_spent = self.get_today_spending()
        daily_limit = self.config.daily_usd_cap

        if today_spent + estimated_cost > daily_limit:
            if today_spent >= daily_limit:
                return BudgetResult.EXCEEDED
            else:
                return BudgetResult.WARNING

        return BudgetResult.OK
```

Workflow Engine

Auto-Merge Workflow

```
┌─────────────┐
│ Code Change │
└──────┬──────┘
       │
       ▼
┌─────────────────┐
│  Run Fast Tests │
└──────┬──────────┘
       │
   ┌───┴───┐
   │       │
   ▼       ▼
┌──────┐ ┌──────┐
│GREEN │ │ RED  │
└──┬───┘ └───┬──┘
   │         │
   ▼         ▼
┌──────────┐ ┌──────────────┐
│ Promotion│ │ AI Diagnosis │
│   Gate   │ │  + Fix Loop  │
└──┬───────┘ └──────────────┘
   │
   ▼
┌───────────────┐
│Fast-Forward   │
│Merge to Trunk │
└──────┬────────┘
       │
       ▼
┌──────────────┐
│Optional CI   │
│Smoke Tests   │
└──────┬───────┘
       │
   ┌───┴───┐
   │       │
   ▼       ▼
┌──────┐ ┌──────────┐
│GREEN │ │   RED    │
│Done! │ │Auto-     │
│      │ │Rollback  │
└──────┘ └──────────┘
```

Promotion Gate

Gate Checks (`sologit/workflows/promotion_gate.py`):

```
class PromotionGate:
    def can_promote(self, workpad: Workpad) -> GateResult:
        checks = [
            self.check_tests_passed(workpad),
            self.check_no_conflicts(workpad),
            self.check_trunk_not_moved(workpad),
            self.check_required_approvals(workpad),
            self.check_code_quality(workpad),
        ]

        failed = [c for c in checks if not c.passed]

        if failed:
            return GateResult.BLOCKED(reasons=failed)

        return GateResult.APPROVED
```

Configurable Rules:

```
promotion_gate:
  required:
    - all_tests_passed
    - no_merge_conflicts

  optional:
    - code_coverage_threshold: 80
    - no_lint_errors
    - security_scan_passed

  allow_override: true  # Manual force-promote
```

Data Flow

End-to-End Request Flow

User Prompt → Merged Code:

```
1. User Input (CLI/TUI/GUI)
   │
   ▼
2. Command Parser
   │
   ▼
3. State Manager (load current state)
   │
   ▼
4. AI Orchestration
   │
   ├─> Task Classification
   ├─> Model Selection
   ├─> Cost Check
   ├─> API Request (Abacus.ai)
   └─> Response Processing
   │
   ▼
5. Patch Engine
   │
   ├─> Generate Unified Diff
   ├─> Validate Patch
   └─> Apply to Workpad
   │
   ▼
6. Test Orchestrator
   │
   ├─> Create Sandbox
   ├─> Run Tests (parallel)
   ├─> Aggregate Results
   └─> Clean up Sandbox
   │
   ▼
7. Promotion Gate
   │
   ├─> Check Tests
   ├─> Check Conflicts
   └─> Check Rules
   │
   ▼
8. Git Engine
   │
   ├─> Fast-Forward Merge
   └─> Update Trunk
   │
   ▼
9. State Manager (save new state)
   │
   ▼
10. UI Update (notify user)
```

Event Flow:

```
# Internal event bus
event_bus.emit(
    Event.WORKPAD_CREATED,
    WorkpadCreatedEvent(id=pad_id, title="add-auth")
)

event_bus.emit(
    Event.TESTS_STARTED,
    TestsStartedEvent(workpad_id=pad_id)
)

event_bus.emit(
    Event.TESTS_PASSED,
    TestsPassedEvent(workpad_id=pad_id, coverage=0.85)
)

event_bus.emit(
    Event.WORKPAD_PROMOTED,
    PromotionEvent(workpad_id=pad_id, commit_sha="abc123")
)
```

Integration Points

Abacus.ai RouteLLM API

Authentication:

```
headers = {
    "Authorization": f"Bearer {api_key}",
    "Content-Type": "application/json"
}
```

Request Format:

```
request = {
    "model": "gpt-4o",
    "messages": [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_prompt}
    ],
    "temperature": 0.2,
    "max_tokens": 4096,
    "stream": False
}

response = requests.post(
    f"{endpoint}/chat/completions",
    headers=headers,
    json=request
)
```

Model Endpoints:
- Planning: `gpt-4o`, `claude-3-5-sonnet`

- Coding: `deepseek-coder-33b` , `codellama-70b-instruct`
- Fast: `llama-3.1-8b-instruct` , `gemma-2-9b-it`

Jenkins Integration (Optional)

Jenkinsfile (example):

```
pipeline {
    agent any

    stages {
        stage('Smoke Tests') {
            steps {
                sh 'pytest tests/smoke/ --quiet'
            }
        }
    }

    post {
        failure {
            script {
                // Trigger Solo Git rollback
                sh 'evogitctl rollback --last'
            }
        }
    }
}
```

API Integration:

```python
class JenkinsOrchestrator:
    def trigger_build(self, job_name: str) -> BuildResult:
        """Trigger Jenkins job"""

    def get_build_status(self, job_name: str, build_id: int) -> BuildStatus:
        """Poll for build status"""

    def handle_build_failure(self, build: Build) -> None:
        """Auto-rollback on CI failure"""
```

Security & Privacy

Data Security

1. **API Keys**: Stored in environment variables or encrypted config
2. **Code Privacy**: All code processing happens locally; only prompts sent to API
3. **Audit Logs**: All AI operations logged with hashes for verification
4. **No Data Collection**: Solo Git collects no telemetry

Safe Operations

1. **Atomic Transactions**: All Git operations are atomic
2. **Rollback Safety**: Every commit can be rolled back
3. **Sandbox Isolation**: Tests run in isolated subprocesses

4. **No Force-Push**: Fast-forward merges only prevent history rewriting

Performance Considerations

Optimization Strategies

1. **Parallel Test Execution**: Run independent tests concurrently
2. **Incremental State Updates**: Only sync changed state
3. **Lazy Loading**: Load repository data on demand
4. **Model Caching**: Cache AI responses for identical prompts
5. **Diff-Based Sync**: Only transfer changed files to GUI

Performance Metrics

- **Workpad Creation**: < 100ms
- **Patch Application**: < 500ms
- **Fast Tests**: < 30s target
- **AI Planning**: 4-10s typical
- **AI Coding**: 5-15s typical
- **Auto-Merge**: < 1s

Scalability

- **Repository Size**: Tested up to 100K files
- **Concurrent Workpads**: Up to 50 active workpads
- **Test Parallelism**: Up to 8 concurrent test processes
- **State File Size**: Typically < 1MB

Future Architecture Considerations

Phase 5 Roadmap

1. **Local Model Support**: Ollama integration for offline operation
2. **Distributed State**: Support for team collaboration
3. **Plugin System**: Extensible architecture for custom workflows
4. **Real-Time Collaboration**: WebSocket-based live updates
5. **Advanced Caching**: Response caching with semantic similarity

Scalability Improvements

1. **Database Backend**: SQLite for large repositories
2. **Streaming State**: Incremental state updates via WebSocket
3. **CDN for Assets**: Offload GUI assets to CDN
4. **Distributed Testing**: Cloud-based test execution

Conclusion

Solo Git’s architecture prioritizes:
- **Simplicity**: Clear component boundaries
- **Reliability**: Fail-safe operations, comprehensive testing
- **Performance**: Parallel execution, optimized data flow
- **Extensibility**: Plugin-ready design
- **User Experience**: Three seamless interface modes

For implementation details, see:
- PROJECT_STRUCTURE.md (PROJECT_STRUCTURE.md) - File organization
- docs/API.md (docs/API.md) - API reference
- docs/HEAVEN_INTERFACE.md (docs/HEAVEN_INTERFACE.md) - UI specifications