- SPEC-DOC-005: Development & Contribution Guide
    - Objectives
    - Scope
        - In Scope
        - Out of Scope
    - Deliverables
    - Success Criteria
    - Related SPECs
- Adding Slash Commands
    - Command Registry Pattern
    - Step-by-Step Guide
        - 1. Define Command Enum
        - 2. Add to Registry
        - 3. Create Handler
        - 4. Wire to Routing
        - 5. Add Tests
        - 6. Add Documentation
    - Example: Complete Command
        - 1. Enum
        - 2. Parser
        - 3. Handler
        - 4. Routing
        - 5. Test
    - Summary
- Build System
    - Cargo Profiles
        - dev-fast (Default Development)
        - dev (Standard Debug)
        - release (Production)
        - perf (Performance Testing)
    - Build Flags
        - TRACE_BUILD
        - DETERMINISTIC
    - Cross-Compilation
        - Linux → macOS
        - Linux → Windows
    - Workspace Structure
    - Summary
- Code Style Guide
    - rustfmt (Formatting)
        - Configuration
        - Format Code
        - Check Formatting
    - Clippy (Linting)
        - Run Clippy
        - Common Clippy Fixes
    - Code Guidelines
        - Naming Conventions
        - Documentation
        - Error Handling
    - Allowed Lints
    - Summary
- Debugging Guide
    - Logging
        - Enable Rust Logging
        - API Request Logging
    - Tmux Sessions
        - View Active Sessions
        - Attach to Session

        - Kill Session
    - MCP Debugging
        - MCP Inspector
        - MCP Logs
    - Agent Debugging
        - Agent Spawn Failures
        - Consensus Issues
    - Debugger (LLDB/GDB)
        - VS Code
        - CLI (LLDB)
    - Performance Debugging
        - Profiling
        - Memory Leaks
    - Common Issues
        - Build Fails
        - Tests Fail
        - Slow Performance
    - Summary
- Development Environment Setup
    - Prerequisites
        - System Requirements
    - Required Tools
        - 1. Rust Toolchain
        - 2. Node.js & npm
        - 3. Git
    - Optional Tools
        - 4. Development Tools
    - Clone Repository
    - Build Project
        - Quick Build (Fast Profile)
        - Full Build (Release Profile)
        - Verify Build
    - Run Tests
        - All Tests
        - Fast Tests (Curated)
        - Specific Module
    - MCP Server Setup (Optional)
        - Local-Memory MCP
        - Filesystem MCP
    - IDE Setup
        - VS Code
        - IntelliJ IDEA / CLion
    - Environment Variables
        - Required for Development
        - Optional for Testing
    - Verify Setup
        - Checklist
    - Troubleshooting
        - Build Errors
        - Test Failures
        - Slow Builds
    - Summary
- Git Workflow
    - Branching Strategy
        - Main Branch
        - Feature Branches
    - Conventional Commits
        - Format
        - Types
        - Examples

- Commit Best Practices
- Pull Request Process
  - 1. Create PR
  - 2. PR Template
  - 3. Review Process
  - 4. Merge
- Upstream Sync
- Summary

Pre-Commit Hooks Guide
- Setup
  - Install Hooks
- Hook: Pre-Commit
- Hook: Pre-Push
- Bypass Hooks (Emergency Only)
  - Skip Pre-Commit
  - Skip Pre-Push
- Debugging Hooks
  - Manual Run
  - Verbose Output
- Common Issues
- Summary

Release Process
- Versioning
- Release Workflow
  - 1. Prepare Release
  - 2. Tag Release
  - 3. GitHub Actions
  - 4. Verify Release
- Homebrew Formula
- Changelog Generation
- Release Checklist
- Summary

Upstream Sync Process
- Overview
- Process
  - 1. Add Upstream Remote
  - 2. Fetch Upstream
  - 3. Merge Upstream
  - 4. Resolve Conflicts
  - 5. Test After Merge
  - 6. Commit and Push
- Conflict Minimization
- Summary

SPEC-DOC-005: Development & Contribution Guide

Status: Pending **Priority**: P1 (Medium) **Estimated Effort**: 10-14 hours **Target Audience**: Contributors, maintainers **Created**: 2025-11-17

Objectives

Provide complete guide for developers contributing to the project: 1. Development environment setup 2. Build system (profiles, fast builds, cross-compilation) 3. Git workflow (branching, commits, PR process) 4. Code style (rustfmt, clippy, lints) 5. Pre-commit hooks (setup, bypass, debugging) 6. Upstream sync process (quarterly merge) 7. Adding new commands (command registry, routing, handlers) 8. Debugging guide (logs, tmux, MCP, agent issues) 9. Release process (versioning, changelog, Homebrew)

Scope

In Scope

- Dev environment setup (Rust toolchain, Node.js, MCP servers)
- Build system (Cargo profiles: dev-fast, release, perf)
- Git workflow (conventional commits, branching strategy)
- Code style enforcement (rustfmt, clippy –all-targets –all-features)
- Pre-commit hooks (setup-hooks.sh, .githooks/)
- Upstream sync (quarterly merge, conflict resolution, UPSTREAM-SYNC.md)
- Adding slash commands (command registry pattern)
- Debugging techniques (logs, tmux sessions, MCP debugging)
- Release process (versioning, changelog generation, Homebrew formula)

Out of Scope

- Architecture details (see SPEC-DOC-002)
- Testing guidelines (see SPEC-DOC-004)
- User-facing documentation (see SPEC-DOC-001)

Deliverables

1. **content/development-setup.md** - Environment, dependencies, tools
2. **content/build-system.md** - Cargo profiles, fast builds, cross-compilation
3. **content/git-workflow.md** - Branching, commits, PRs, conventional commits
4. **content/code-style.md** - rustfmt, clippy, lints, guidelines
5. **content/pre-commit-hooks.md** - Setup, debugging, bypass
6. **content/upstream-sync.md** - Quarterly merge process
7. **content/adding-commands.md** - Command registry, routing, examples
8. **content/debugging-guide.md** - Logs, tmux, MCP, agents
9. **content/release-process.md** - Versioning, changelog, publishing

---

Success Criteria

- [ ] New contributor can set up dev environment in 30 minutes
- [ ] Build system documented with all profiles
- [ ] Git workflow clearly explained

- [ ] Pre-commit hooks setup guide complete
- [ ] Adding commands tutorial with working example
- [ ] Debugging techniques comprehensive

Related SPECs

- SPEC-DOC-000 (Master)
- SPEC-DOC-002 (Core Architecture - for deep understanding)
- SPEC-DOC-004 (Testing - for testing contributions)

Status: Structure defined, content pending

Adding Slash Commands

Guide to adding new `/command` to the spec-kit framework.

Command Registry Pattern

Location: `codex-rs/tui/src/chatwidget/spec_kit/command_registry.rs`

Pattern: Command registry maps `/command` → handler function

Step-by-Step Guide

1. Define Command Enum

File: `command_registry.rs`

```
#[derive(Debug, Clone, PartialEq)]
pub enum SpecKitCommand {
    // Existing commands
    New,
    Plan,
    Status,
    // Add your command
    MyNewCommand { arg1: String },
}
```

---

2. Add to Registry

```
pub fn parse_speckit_command(input: &str) -> Option<SpecKitCommand>
{
    if input.starts_with("/speckit.mynew ") {
        let args = input.strip_prefix("/speckit.mynew ")?.trim();
        return Some(SpecKitCommand::MyNewCommand {
            arg1: args.to_string()
        });
    }
```

```
        // ... existing commands
        None
    }
```

---

3. Create Handler

File: `command_handlers.rs`

```
    pub fn handle_my_new_command(
        spec_id: &str,
        config: &SpecKitConfig,
    ) -> Result<String> {
        // Implementation
        let result = do_something(spec_id)?;

        // Return formatted response
        Ok(format!("Command executed: {}", result))
    }
```

---

4. Wire to Routing

File: `routing.rs` (or main handler)

```
    match command {
        SpecKitCommand::MyNewCommand { arg1 } => {
            handle_my_new_command(&arg1, config)?
        }
        // ... existing commands
    }
```

---

5. Add Tests

File: `command_registry_tests.rs`

```
    #[test]
    fn test_parse_mynew_command() {
        let input = "/speckit.mynew test-arg";
        let cmd = parse_speckit_command(input);

        assert_eq!(
            cmd,
            Some(SpecKitCommand::MyNewCommand {
                arg1: "test-arg".to_string()
            })
        );
    }

    #[test]
    fn test_handle_mynew_command() {
        let result = handle_my_new_command("SPEC-TEST",
&default_config());
        assert!(result.is_ok());
    }
```

---

6. Add Documentation

Update: `docs/SPEC-DOC-003/content/command-reference.md`

```
### /speckit.mynew

**Purpose**: Brief description

**Usage**:
\`\`\`bash
/speckit.mynew <arg>
\`\`\`

**Example**:
\`\`\`bash
/speckit.mynew test-value
\`\`\`

**Output**: Description of output
```

Example: Complete Command

Command: `/speckit.hello <name>`

1. Enum

```rust
pub enum SpecKitCommand {
    Hello { name: String },
}
```

2. Parser

```rust
if input.starts_with("/speckit.hello ") {
    let name = input.strip_prefix("/speckit.hello ")?.trim().to_string();
    return Some(SpecKitCommand::Hello { name });
}
```

3. Handler

```rust
pub fn handle_hello(name: &str) -> Result<String> {
    Ok(format!("Hello, {}!", name))
}
```

4. Routing

```rust
SpecKitCommand::Hello { name } => {
    handle_hello(&name)?
}
```

5. Test

```rust
#[test]
fn test_hello_command() {
    let result = handle_hello("World");
    assert_eq!(result.unwrap(), "Hello, World!");
}
```

Summary

Steps: 1. Add to command enum 2. Parse in registry 3. Create handler 4. Wire to routing 5. Add tests 6. Update docs

Files Modified: - `command_registry.rs` - `command_handlers.rs` - `routing.rs` (or handler.rs) - `*_tests.rs`

Next: Debugging Guide

Build System

Comprehensive guide to the Cargo build system and profiles.

Cargo Profiles

dev-fast (Default Development)

Purpose: Fast incremental builds for local development

Build Time: ~30-60s (incremental: ~5-10s)

Command:

```
./build-fast.sh
# Or: cargo build --profile dev-fast
```

Output: `codex-rs/target/dev-fast/code`

Optimizations: - opt-level = 1 (basic optimizations) - debug = false (no debug symbols) - incremental = true

---

dev (Standard Debug)

Purpose: Full debug info for debugging

Build Time: ~2-5 minutes

Command:

```
cargo build
```

Output: `codex-rs/target/debug/code`

Optimizations: - opt-level = 0 - debug = true - incremental = true

---

release (Production)

Purpose: Optimized for production

Build Time: ~5-10 minutes

Command:

```
cargo build --release
```

Output: `codex-rs/target/release/code`

Optimizations: - opt-level = 3 - lto = true (link-time optimization) - codegen-units = 1

---

perf (Performance Testing)

Purpose: Profiling and benchmarking

Command:

```
./build-fast.sh perf
```

Optimizations: - opt-level = 3 - debug = true (for profiling symbols)

Build Flags

TRACE_BUILD

Purpose: Print build metadata

```
TRACE_BUILD=1 ./build-fast.sh
```

Output: Toolchain version, artifact SHA

DETERMINISTIC

Purpose: Reproducible builds

```
DETERMINISTIC=1 ./build-fast.sh
```

Behavior: Removes timestamps, UUIDs

Cross-Compilation

Linux → macOS

```
rustup target add x86_64-apple-darwin
cargo build --target x86_64-apple-darwin --release
```

Linux → Windows

```
rustup target add x86_64-pc-windows-gnu
cargo build --target x86_64-pc-windows-gnu --release
```

Workspace Structure

Root: `codex-rs/Cargo.toml`

Packages: - codex-tui (main TUI) - codex-core (conversation logic) - codex-cli (CLI entry point) - mcp-client (MCP integration) - 20+ other crates

Build All:

```
cd codex-rs
cargo build --workspace
```

Summary

Profiles: - dev-fast: Fast dev builds (~30-60s) - dev: Full debug (~2-5min) - release: Production (~5-10min) - perf: Profiling (~5-10min)

Next: Git Workflow

Code Style Guide

Rust code style, formatting, and linting guidelines.

---

rustfmt (Formatting)

Configuration

File: `codex-rs/rustfmt.toml`

Key Settings: - Edition: 2024 - Max width: 100 - Tab spaces: 4

Format Code

```
cd codex-rs
cargo fmt --all
```

Check Formatting

```
cargo fmt --all -- --check
```

Pre-commit hook: Automatically runs format check

Clippy (Linting)

Run Clippy

```
cd codex-rs
cargo clippy --workspace --all-targets --all-features -- -D warnings
```

Flags: - `--all-targets`: Check tests, benches, examples - `--all-features`: Check all feature combinations - `-D warnings`: Treat warnings as errors

Common Clippy Fixes

Unused imports:

```rust
// Bad
use std::collections::HashMap;

// Good (if unused, remove)
```

Unnecessary clones:

```rust
// Bad
let s = string.clone();

// Good (if ownership not needed)
let s = &string;
```

Code Guidelines

Naming Conventions

Functions: snake_case

```rust
fn calculate_total() { }
```

Types: PascalCase

```rust
struct UserAccount { }
enum RequestStatus { }
```

Constants: SCREAMING_SNAKE_CASE

```rust
const MAX_RETRIES: usize = 3;
```

Documentation

Public APIs:

```rust
/// Calculates the total cost with tax
///
/// # Arguments
/// * `subtotal` - Base amount before tax
/// * `tax_rate` - Tax rate (0.0-1.0)
///
/// # Returns
/// Total amount including tax
pub fn calculate_total(subtotal: f64, tax_rate: f64) -> f64 {
    subtotal * (1.0 + tax_rate)
}
```

Error Handling

Use Result:

```rust
// Good
fn parse_config(path: &Path) -> Result<Config> {
    let contents = fs::read_to_string(path)?;
    let config: Config = toml::from_str(&contents)?;
    Ok(config)
}

// Bad
fn parse_config(path: &Path) -> Config {
    let contents = fs::read_to_string(path).unwrap(); // ✗
    toml::from_str(&contents).unwrap() // ✗
}
```

Allowed Lints

workspace (Cargo.toml):

```toml
[workspace.lints.clippy]
unwrap_used = "warn"
expect_used = "warn"
panic = "warn"
```

Override in tests:

```rust
#[cfg(test)]
mod tests {
    #![allow(clippy::unwrap_used)]
    // Tests can use .unwrap()
}
```

Summary

Format: `cargo fmt --all` **Lint**: `cargo clippy --workspace --all-targets --all-features -- -D warnings` **Conventions**: snake_case functions, PascalCase types, document public APIs

Next: Pre-Commit Hooks

Debugging Guide

Comprehensive debugging techniques for development.

Logging

Enable Rust Logging

```bash
export RUST_LOG=debug
./codex-rs/target/dev-fast/code
```

Levels: - `error`: Errors only - `warn`: Warnings + errors - `info`: Info + warn + errors - `debug`: Debug + info + warn + errors - `trace`: All messages

Module-specific:

```
export RUST_LOG=codex_tui::chatwidget::spec_kit=debug
```

API Request Logging

```
./codex-rs/target/dev-fast/code --debug
```

Output: `~/.code/debug.log` (API requests/responses)

Tmux Sessions

View Active Sessions

```
tmux ls
```

Example Output:

```
speckit-SPEC-TEST-001-plan: 1 windows (created Fri Nov 17)
```

Attach to Session

```
tmux attach -t speckit-SPEC-TEST-001-plan
```

Detach: `Ctrl-b d`

Kill Session

```
tmux kill-session -t speckit-SPEC-TEST-001-plan
```

MCP Debugging

MCP Inspector

Install:

```
npm install -g @modelcontextprotocol/inspector
```

Use:

```
npx @modelcontextprotocol/inspector npx -y
@modelcontextprotocol/server-memory
```

Features: - Test tool calls - Inspect responses - Debug connection issues

---

MCP Logs

Enable verbose logging:

```
# ~/.code/config.toml
[mcp_servers.local-memory]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-memory", "--verbose"]
```

Agent Debugging

Agent Spawn Failures

Symptoms: Agent doesn't start, timeout errors

Debug:

```
# Check agent availability
claude --version
gemini --version

# Check config
cat ~/.code/config.toml | grep -A 5 "\[agents\]"

# Manual test
claude "test message"
```

Consensus Issues

Symptoms: Empty consensus, degraded mode

Debug:

```
# Check consensus artifacts
ls -la docs/SPEC-OPS-004*/evidence/consensus/SPEC-TEST/

# Inspect consensus file
cat docs/.../consensus/SPEC-TEST/spec-plan_*.json | jq
```

Debugger (LLDB/GDB)

VS Code

.vscode/launch.json:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "lldb",
      "request": "launch",
      "name": "Debug code",
      "cargo": {
        "args": ["build", "--bin=code", "--package=codex-cli"]
      },
      "args": [],
      "cwd": "${workspaceFolder}"
    }
  ]
}
```

Set breakpoint: Click left margin in source file

Run: F5

CLI (LLDB)

```
# Build with debug symbols
cargo build --bin code

# Run in debugger
lldb ./target/debug/code

# Set breakpoint
(lldb) breakpoint set --name main
(lldb) run
```

Performance Debugging

Profiling

```
cargo install flamegraph
cargo flamegraph --bin code
open flamegraph.svg
```

Memory Leaks

```
# macOS
leaks --atExit -- ./target/debug/code

# Linux (valgrind)
valgrind --leak-check=full ./target/debug/code
```

Common Issues

Build Fails

Check:

```
cargo clean
cargo build
```

Tests Fail

Isolate:

```
cargo test --package codex-tui specific_test -- --nocapture
```

Slow Performance

Profile:

```
cargo flamegraph --bin code
```

Summary

Tools: - RUST_LOG (logging) - –debug (API logs) - tmux (session debugging) - MCP inspector (MCP debugging) - lldb/gdb (breakpoints) - flamegraph (profiling)

Next: Release Process

Development Environment Setup

Complete guide to setting up your development environment.

Prerequisites

System Requirements

Minimum: - CPU: 2 cores - RAM: 4 GB - Disk: 2 GB free space - OS: Linux, macOS, or Windows (WSL2)

Recommended: - CPU: 4+ cores - RAM: 8+ GB - Disk: 5 GB free space - OS: Linux or macOS (for best performance)

Required Tools

1. Rust Toolchain

Version: 1.90.0 (Rust Edition 2024)

Install via rustup:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Set version:

```
rustup toolchain install 1.90.0
rustup default 1.90.0
```

Verify:

```
rustc --version
# Should output: rustc 1.90.0 (...)
```

---

2. Node.js & npm

Version: Node.js 20+ (for npm packaging and CLI tooling)

Install:

```
        # Using nvm (recommended)
        curl -o- https://raw.githubusercontent.com/nvm-
sh/nvm/v0.39.0/install.sh | bash
        nvm install 20
        nvm use 20

        # Or via package manager
        # macOS: brew install node@20
        # Ubuntu: sudo apt install nodejs npm
```

Verify:

```
        node --version  # v20.x.x
        npm --version   # 10.x.x
```

---

3. Git

Version: 2.30+

Install:

```
        # macOS
        brew install git

        # Ubuntu
        sudo apt install git

        # Verify
        git --version  # git version 2.x.x
```

Optional Tools

4. Development Tools

cargo-watch (auto-rebuild on changes):

```
        cargo install cargo-watch
```

cargo-tarpaulin (coverage):

```
        cargo install cargo-tarpaulin
```

cargo-flamegraph (profiling):

```
        cargo install flamegraph
```

hyperfine (CLI benchmarking):

```
        cargo install hyperfine
```

Clone Repository

```
# Clone
git clone https://github.com/theturtlecsz/code.git
cd code

# Set up git hooks
bash scripts/setup-hooks.sh

# Verify hooks
git config core.hooksPath
# Should output: .githooks
```

Build Project

Quick Build (Fast Profile)

```
./build-fast.sh
```

Output: `codex-rs/target/dev-fast/code`

Profile: Optimized for fast builds (~30-60s)

Full Build (Release Profile)

```
cd codex-rs
cargo build --release
```

Output: `codex-rs/target/release/code`

Profile: Optimized for performance (~5-10min first build)

Verify Build

```
./codex-rs/target/dev-fast/code --version
# Output: code x.x.x

./codex-rs/target/dev-fast/code --help
# Shows help
```

Run Tests

All Tests

```
cd codex-rs
cargo test --workspace --all-features
```

Time: ~10-15 minutes (604 tests)

Fast Tests (Curated)

```
bash scripts/ci-tests.sh
```

Time: ~3-5 minutes

Specific Module

```
cd codex-rs
cargo test -p codex-tui
```

MCP Server Setup (Optional)

Local-Memory MCP

Purpose: Spec-kit consensus storage

Install:

```
npm install -g @modelcontextprotocol/server-memory
```

Configure (`~/.code/config.toml`):

```
[mcp_servers.local-memory]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-memory"]
startup_timeout_sec = 10
```

Verify:

```
npx -y @modelcontextprotocol/server-memory --version
```

Filesystem MCP

Purpose: File operations via MCP

Configure:

```
[mcp_servers.filesystem]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem",
"/path/to/project"]
```

IDE Setup

VS Code

Extensions: - rust-analyzer (rust-lang.rust-analyzer) - CodeLLDB (vadimcn.vscode-lldb) - Debugging - Even Better TOML (tamasfe.even-better-toml) - Error Lens (usernamehw.errorlens)

Settings (.vscode/settings.json):

```
{
  "rust-analyzer.cargo.features": "all",
  "rust-analyzer.check.command": "clippy",
  "rust-analyzer.check.extraArgs": ["--all-targets", "--all-
features"],
  "editor.formatOnSave": true,
  "[rust]": {
    "editor.defaultFormatter": "rust-lang.rust-analyzer"
  }
}
```

IntelliJ IDEA / CLion

Plugins: - Rust (JetBrains) - TOML (JetBrains)

Settings: - Enable “Run clippy on save” - Enable “Format on save”

Environment Variables

Required for Development

```
# .env or ~/.bashrc

# OpenAI API key (for testing)
export OPENAI_API_KEY=sk-...

# Optional: Logging
export RUST_LOG=info

# Optional: Faster linking (macOS)
export CARGO_PROFILE_DEV_BUILD_OVERRIDE_DEBUG=true
```

Optional for Testing

```
# HAL testing (optional)
export HAL_SECRET_KAVEDARR_API_KEY=...

# Skip HAL tests
export SPEC_OPS_HAL_SKIP=1

# Enable telemetry capture
export SPEC_OPS_TELEMETRY_HAL=1

# Fast test mode (skip some pre-commit checks)
export PRECOMMIT_FAST_TEST=0
```

Verify Setup

Checklist

```
# ✅ Rust toolchain
rustc --version | grep "1.90"

# ✅ Cargo works
```

```
cargo --version

# ✅ Node.js/npm
node --version | grep "v20"

# ✅ Git configured
git config user.name
git config user.email

# ✅ Hooks installed
git config core.hooksPath | grep ".githooks"

# ✅ Build succeeds
./build-fast.sh && ./codex-rs/target/dev-fast/code --version

# ✅ Tests pass
cd codex-rs && cargo test -p codex-login --test all

# ✅ Clippy passes
cargo clippy --workspace --all-targets --all-features -- -D warnings

# ✅ Format check
cargo fmt --all -- --check
```

All checks should pass ✅

Troubleshooting

Build Errors

Error: `rustc version 1.x.x is too old`

```
# Solution: Update Rust
rustup update
rustup default 1.90.0
```

Error: `linker 'cc' not found`

```
# Solution: Install build tools
# macOS: xcode-select --install
# Ubuntu: sudo apt install build-essential
```

Test Failures

Error: Tests fail with "Connection refused"

```
# Solution: MCP server not running (expected if not configured)
# Tests should pass with SPEC_OPS_HAL_SKIP=1
export SPEC_OPS_HAL_SKIP=1
cargo test
```

Slow Builds

Solution 1: Use dev-fast profile

```
./build-fast.sh  # ~30-60s
```

Solution 2: Enable incremental compilation

```
export CARGO_INCREMENTAL=1
```

Solution 3: Use sccache (build cache)

```
cargo install sccache
export RUSTC_WRAPPER=sccache
```

Summary

Setup Time: ~30 minutes

Steps: 1. ✅ Install Rust 1.90.0 2. ✅ Install Node.js 20+ 3. ✅ Clone repository 4. ✅ Set up git hooks (`bash scripts/setup-hooks.sh`) 5. ✅ Build project (`./build-fast.sh`) 6. ✅ Run tests (`bash scripts/ci-tests.sh`) 7. ✅ Configure IDE (VS Code recommended)

Next Steps: - Build System - Cargo profiles, cross-compilation - Git Workflow - Branching, commits, PRs - Code Style - rustfmt, clippy, lints

---

References: - Rust installation: https://rustup.rs/ - Project README: `/README.md` - Setup hooks: `scripts/setup-hooks.sh`

Git Workflow

Git branching strategy, commits, and PR process.

Branching Strategy

Main Branch

Branch: `main` **Protection**: Protected, requires PR **Purpose**: Stable production code

---

Feature Branches

Format: `feature/description` or `username/description`

Examples: - `feature/add-dark-mode` - `fix/database-connection` - `docs/api-documentation`

Create:

```
git checkout -b feature/add-dark-mode
```

Conventional Commits

Format

```
<type>(<scope>): <description>

[optional body]

[optional footer]
```

Types

- `feat`: New feature
- `fix`: Bug fix
- `docs`: Documentation
- `test`: Tests
- `refactor`: Code refactoring
- `perf`: Performance improvement
- `chore`: Build/tooling changes

Examples

```
feat(tui): add dark mode toggle
fix(mcp): resolve connection timeout
docs(api): add MCP integration guide
test(spec-kit): add consensus unit tests
```

Commit Best Practices

DO:

```
# Atomic commits
git commit -m "feat(tui): add command history"
git commit -m "test(tui): add history tests"

# Descriptive messages
git commit -m "fix(db): resolve race condition in pool"

# Present tense
git commit -m "add feature" (not "added feature")
```

DON'T:

```
# Vague messages
git commit -m "fix stuff"

# Multiple changes
git commit -m "add feature, fix bug, update docs"
```

Pull Request Process

1. Create PR

```bash
# Push branch
git push -u origin feature/add-dark-mode

# Create PR (via GitHub UI)
```

2. PR Template

```markdown
## Summary
Add dark mode toggle to TUI settings

## Changes
- Add dark mode theme
- Add toggle in settings
- Update color scheme

## Testing
- Tested on Linux, macOS
- All tests passing

## Checklist
- [x] Tests added
- [x] Documentation updated
- [x] Clippy passing
```

3. Review Process

- CI must pass (tests, clippy, fmt)
- At least 1 approval required
- Address review comments
- Squash/rebase if requested

4. Merge

- Squash and merge (default)
- Delete branch after merge

Upstream Sync

Frequency: Quarterly

Process: See Upstream Sync Guide

Summary

Workflow: 1. Create feature branch 2. Make atomic commits (conventional format) 3. Push and create PR 4. Pass CI + review 5. Squash and merge

Next: Code Style

Pre-Commit Hooks Guide

Setup, debugging, and bypass procedures for git hooks.

Setup

Install Hooks

```
bash scripts/setup-hooks.sh
```

Verifies:

```
git config core.hooksPath
# Output: .githooks
```

Hook: Pre-Commit

Location: `.githooks/pre-commit`

Runs: Policy compliance checks (< 5s)

Checks: 1. Storage policy (local-memory usage) 2. Tag schema (namespacing)

Trigger: Only runs if spec_kit files modified

Hook: Pre-Push

Runs: Format, lint, build checks (~2-5min)

Checks: 1. `cargo fmt --all -- --check` 2. `cargo clippy --workspace --all-targets --all-features -- -D warnings` 3. `cargo build --workspace --all-features`

Bypass Hooks (Emergency Only)

Skip Pre-Commit

```
git commit --no-verify -m "Emergency hotfix"
```

Skip Pre-Push

```
PREPUSH_FAST=0 git push
```

Use sparingly: Only for emergencies

Debugging Hooks

Manual Run

```
# Pre-commit
bash .githooks/pre-commit

# Specific check
bash scripts/validate_storage_policy.sh
```

Verbose Output

```
# Enable debug
set -x
bash .githooks/pre-commit
```

Common Issues

Issue: Hook doesn't run

Solution:

```
git config core.hooksPath
# If not .githooks, re-run setup
bash scripts/setup-hooks.sh
```

Issue: Hook fails on unrelated files

Solution: Hooks only run for spec_kit changes. Check modified files:

```
git diff --cached --name-only | grep spec_kit
```

Summary

Setup: `bash scripts/setup-hooks.sh` **Bypass**: `git commit --no-verify` (emergencies only) **Debug**: Run hooks manually

Next: Upstream Sync

Release Process

Versioning, changelog, and publishing workflow.

Versioning

Scheme: Semantic Versioning (SemVer)

Format: `MAJOR.MINOR.PATCH`

- MAJOR: Breaking changes
- MINOR: New features (backward compatible)
- PATCH: Bug fixes

Release Workflow

1. Prepare Release

Update version (`codex-cli/package.json`):

```json
{
  "version": "1.2.3"
}
```

Update Changelog (`CHANGELOG.md`):

```markdown
## [1.2.3] - 2025-11-17

### Added
- Dark mode support

### Fixed
- Database connection timeout

### Changed
- Improved error messages
```

---

2. Tag Release

```bash
git tag -a v1.2.3 -m "Release v1.2.3"
git push origin v1.2.3
```

---

3. GitHub Actions

Triggers: Push to `main` or tag push

Jobs: 1. Build (Linux, macOS, Windows) 2. Test (all platforms) 3. Publish to npm

Workflow: `.github/workflows/release.yml`

---

4. Verify Release

npm:

```bash
npm view @just-every/code version
# Should show: 1.2.3
```

GitHub: - Check release notes - Verify binaries attached

Homebrew Formula

Update formula (`homebrew-tap/Formula/code.rb`):

```ruby
class Code < Formula
  desc "Fast local coding agent"
  homepage "https://github.com/theturtlecsz/code"
  version "1.2.3"
  # ... download URLs, SHA256
end
```

Generate:

```
bash scripts/generate-homebrew-formula.sh v1.2.3
```

Changelog Generation

Manual:

```
## [1.2.3] - 2025-11-17

### Added
- List new features

### Fixed
- List bug fixes

### Changed
- List changes
```

Automated (future):

```
# Generate from git commits
git-cliff --tag v1.2.3 > CHANGELOG.md
```

Release Checklist

- [ ] Update version in package.json
- [ ] Update CHANGELOG.md
- [ ] Run full test suite (`cargo test --workspace`)
- [ ] Build release (`cargo build --release`)
- [ ] Create git tag (`git tag -a v1.2.3`)
- [ ] Push tag (`git push origin v1.2.3`)
- [ ] Verify CI passes
- [ ] Check npm publish
- [ ] Update Homebrew formula
- [ ] Create GitHub release notes

Summary

Process: 1. Update version + changelog 2. Tag release 3. Push (CI auto-publishes) 4. Update Homebrew formula 5. Verify release

Workflow: `.github/workflows/release.yml`

References: - SemVer: https://semver.org/ - Changelog: `CHANGELOG.md`

Upstream Sync Process

Quarterly merge process for upstream changes.

Overview

Upstream: https://github.com/just-every/code **Frequency**: Quarterly (or as needed) **Strategy**: Merge with manual conflict resolution

---

Process

1. Add Upstream Remote

```
git remote add upstream https://github.com/just-every/code.git
git remote -v
# upstream  https://github.com/just-every/code.git (fetch)
```

---

2. Fetch Upstream

```
git fetch upstream
git fetch upstream --tags
```

---

3. Merge Upstream

```
# Create merge commit (no fast-forward)
git merge --no-ff --no-commit upstream/main

# Review conflicts
git status
```

---

4. Resolve Conflicts

Isolation Strategy: Fork-specific code in `tui/src/chatwidget/spec_kit/`

Conflict Resolution: - Accept upstream changes for non-spec_kit files - Keep fork changes for spec_kit files - Manually merge if both modified same file

Example:

```
# spec_kit conflict - keep ours
git checkout --ours codex-rs/tui/src/chatwidget/spec_kit/handler.rs

# Upstream change - keep theirs
git checkout --theirs codex-rs/tui/src/chatwidget/widget.rs
```

---

5. Test After Merge

```
# Build
./build-fast.sh
```

```bash
# Test
bash scripts/ci-tests.sh

# Full test suite
cd codex-rs && cargo test --workspace
```

---

6. Commit and Push

```bash
git add .
git commit -m "chore: merge upstream/main (2025-11-17)"
git push
```

Conflict Minimization

98.2% Isolation Achieved: Spec-kit code isolated in separate modules

Low-Conflict Areas: - `tui/src/chatwidget/spec_kit/*` (fork-specific) - `docs/SPEC-*` (fork-specific) - `.githooks/*` (fork-specific)

High-Conflict Areas (merge carefully): - `Cargo.toml` (dependencies) - `tui/src/chatwidget/widget.rs` (TUI core) - `core/src/*` (conversation logic)

Summary

Frequency: Quarterly **Process**: Fetch → Merge → Resolve → Test → Commit **Strategy**: Keep spec_kit changes, accept upstream otherwise

References: - Upstream sync docs: `docs/UPSTREAM-SYNC.md` - Conflict resolution: `.git/MERGE_HEAD`

Next: Adding Commands
