

Forge

Directed-Graph Multi-Agent CLI for Production-Ready Code

Version 0.6

Python 3.11+

MIT License

github.com/shaheennazir/forge

Table of Contents

1. Project Overview
2. Architecture -- Directed Graph Model
3. The Four Pipeline Nodes
3.1 Orchestrator
3.2 Spec Generator
3.3 Executor (TDD Workflow)
3.4 Review Gate
4. Database -- 4-Tier Memory System
5. LLM Backends and Tool Calling
6. The Product Compiler (9-Stage Pipeline)
7. TUI -- Textual Full-Screen Interface
8. Skills System
9. MCP -- Model Context Protocol
10. File Service and Git-Aware Patch System
11. Subagent Delegation
12. CLI Commands
13. Project Structure
14. Key Design Principles

1. Project Overview

Forge is a CLI tool that builds production-ready code through a directed-graph pipeline. Rather than a single monolithic agent, it uses four purpose-built nodes -- **Orchestrator**, **Spec Generator**, **Executor**, and **Review Gate** -- wired together as a graph. Each node can be retried independently on failure, and the entire project is stored in a 4-tier SQLite memory system (short/mid/episodic/long).

The project targets full-stack developers who want structured, auditable AI-assisted coding. Built with Python 3.11+, it ships with a full Textual TUI (`forge tui`), a Product Compiler for structured product ideation, and deep integrations for OpenAI, Anthropic, MiniMax, and Ollama. The design prioritizes explicit failure lanes -- when the Review Gate says no, the Executor retries with the specific failure reason.

Entry point	src/forgе/cli.py
Package root	src/forgе/
Config directory	~/.forge/ (workspace, skills, mcp_servers.yaml)
Database	SQLite at ~/.forge/projects/<id>/memory.db
Key dependencies	openai, anthropic, structlog, textual, pydantic, pytest
Test suite	~/forge/tests/ -- run with: pytest

What Forge Does

Forge takes a user prompt and transforms it into production-ready code through an auditable, multi-stage pipeline. It does this by modelling each stage of work as a node in a directed graph. Nodes communicate via typed inputs/outputs, failures are explicit and retrieable, and every decision is recorded in a 4-tier memory system.

Two Operational Modes

- **Forge Pipeline (forge run):** Orchestrator decides whether to generate a SPEC.md or go straight to execution. Executor runs TDD cycles. Review Gate checkpoints quality.

- **Product Compiler (forge compile):** A heavier 9-stage pipeline for building complete applications from scratch. Interview → User Flows → API Contracts → Rules → DB Schema → Failing Tests → Implementation - > Integration → Review.

2. Architecture -- Directed Graph Model

Everything in Forge revolves around `ForgeGraph` (`src/forge/graph.py`). The project is modelled as a directed graph where nodes are units of work and edges define the control flow. This differs from a linear pipeline because branches can retry independently and the graph structure makes it easy to inspect state at any node.

NodeStatus Enum -- What Each Node Reports

Status	Meaning
PENDING	Node has not started yet
RUNNING	Currently executing
DONE	Completed successfully
BLOCKED	Review Gate said no -- awaiting retry. The node is alive and can retry.
FAILED	Unrecoverable error
SKIPPED	Not applicable for this run

Graph Storage (graph.py)

```
self._nodes: dict[str, dict]    # node id -> {id, type, status,
inputs, output, error}
self._edges: list[tuple[str, str]] # (source_id, target_id)
self._out_edges: dict[str, list[str]] # node id -> [target_ids]
self._in_degree: dict[str, int]      # unresolved incoming edge
count
```

Key Graph Methods

- `add_node(id, node_type)` -- registers a new node in the graph
- `add_edge(from, to)` -- adds a directed edge between nodes

- `next_ready()` -- returns nodes whose predecessors are all DONE; enables parallel execution
- `update_node_status(id, status, error?)` -- updates node state
- `store_output(id, output)` -- saves execution result to graph node
- `is_complete()` -- True when all non-SKIPPED nodes are DONE
- `get_path_to_root(node_id)` -- traces all predecessors for context injection

How Graph Traversal Works

The orchestrator calls `g.next_ready()` to find which nodes can run -- those with all predecessors in DONE state. When a node completes, its status is updated and `next_ready()` is called again. This enables true parallel sub-agent execution when multiple nodes are ready simultaneously. The traversal loops until `g.is_complete()` returns True.

3. The Four Pipeline Nodes

3.1 Orchestrator (src/forge/runners/orchestrator.py)

The Orchestrator is the brain -- it decides what to do when the user provides a prompt. It runs first and determines whether the project needs a new SPEC.md (via Spec Generator) or direct execution (via Executor).

Line-by-line flow

- Receives the raw user prompt
- Sends it to the LLM with ORCHESTRATOR_SYSTEM prompt (defined in `runners/common.py` line 39)
- LLM returns JSON: `{"action": "generate_spec"|"executor"|"done"|"delegate", "reason": "..."}"`
- If `generate_spec` → adds SpecGen node to graph, sets `current = spec_gen`
- If `executor` → adds Execute node, sets `current = executor`
- If `delegate` → spawns a SubagentManager subagent for complex subtasks
- If `done` → orchestrator exits; graph traversal is complete
- Loops until done; each iteration is one full graph traversal

ORCHESTRATOR_SYSTEM prompt (common.py, line 39)

```
You are the Orchestrator in a directed-graph agentic coding system (forge).  
You receive a user prompt and must decide the next action.
```

```
Available actions:
```

- ```
- generate_spec: User has a vague idea, produce a concrete SPEC.md first
- executor: User has a spec, generate code files
- done: All work is complete
- delegate: Offload a specific subtask to a subagent
```

```
Respond ONLY with JSON: {"action": "...", "reason": "..."}"
```

## 3. The Four Pipeline Nodes (cont.)

---

### 3.2 Spec Generator (runners/spec\_gen.py)

Spec Generator turns a vague request into a complete SPEC.md. First step in any new project. Everything is append-only -- no spec is ever overwritten.

#### Line-by-line flow

- **Line 46:** `db.latest_spec_version()` checks if a spec already exists for this project
- **Lines 47-53:** If existing spec found, loads it as `existing_spec_md` so LLM can update (not replace)
- **Line 56:** `version = (existing_version or 0) + 1` -- version number increments each run
- **Lines 58-83:** LLM is called with skill context injected for "planning" task type
- **Lines 70-75:** If existing spec, prompt reformatted to UPDATE the spec (preserving what's already good)
- **Line 84:** `build_spec_changelog()` generates a human-readable changelog entry
- **Line 89:** `db.save_spec_version()` saves as append-only row in `spec_versions` table
- **Lines 90-95:** Writes `current_spec_version` and `current_spec_md` to mid-tier memory
- **Line 97:** `g.update_node_status(node_id, NodeStatus.DONE)`
- **Lines 98-100:** Stores `{spec_version, spec_md}` as graph node output

#### Skill injection function (line 18-26)

```
def inject_skills_into_context(task_type, skill_registry):
 matched = skill_registry.match(task_type)
 return [f"[skill:{s.name}] {s.description}" for s in matched]
```

Matched skills are appended to the SPEC\_GEN\_SYSTEM prompt so the LLM knows relevant procedural patterns (e.g. how to write a good SPEC.md).

## 3. The Four Pipeline Nodes (cont.)

---

### 3.3 Executor -- TDD Workflow (runners/executor.py)

The Executor reads SPEC.md and generates code using a strict TDD (Test-Driven Development) cycle: tests first, then implementation, then verify.

#### Phase 1 -- Test Generation (lines 88-116)

- Builds TDD prompt asking LLM to generate tests from SPEC.md
- Skill registry matched for "code" and "tdd" task types
- MCP tools context and LSP (code intelligence) context injected into prompt
- `with_retry()` wraps the call with `RetryPolicy(max_attempts=5)` for resilience
- Result parsed as JSON: `[{"path": "tests/...", "action": "create", "content": "..."}]`

#### Phase 2 -- Implementation (lines 118-183)

- Second LLM call generates implementation files
- `FileService.status()` shows what changed since HEAD (git-aware context)
- Current content of top 5 changed text files injected so LLM writes surgical patches
- Prompt explicitly tells LLM: prefer action "patch" over "write"/"update" for existing files
- `RetryPolicy` also applies to this phase

#### Phase 3 -- Test Execution (lines 185-189, `run_tests` in `common.py`)

- `_run_tests_with_write()` writes test files first, then calls `run_tests()`
- `run_tests()`: `pytest --json-report --json-report-file=results.json -q`
- Reads `results.json`; falls back to regex-parsed stdout if `pytest-json-report` not installed
- Returns `{total, failed, passed, output, failures}`

## File Writing and Permissions (lines 191-213)

- `check_permission()` called for `build:write` and `build:delete` before any write operation
- DENY → raises `PermissionError`; ASK → raises `PermissionRequired` (prompts user in TUI)
- `write_files()` in `common.py` handles create/write/update/patch/delete actions
- patch action uses `forge.patch` for surgical edits with context-line anchoring
- Mid-tier memory stores `last_files_generated`, `executor_done_at`, `workdir`

## 3. The Four Pipeline Nodes (cont.)

---

### 3.4 Review Gate (runners/review\_gate.py)

The Review Gate is the quality checkpoint. It evaluates executor output against the spec and decides whether to pass or block the pipeline.

#### Decision Priority

- **Priority 1: test failures** → automatic fail. Extracts file/line/type/message from pytest JSON report
- **Priority 2: LLM review.** If LLM available and files exist, LLM verifies against spec using `REVIEW_SYSTEM` prompt. Must return JSON: `{"pass": bool, "reason": "...", "issues": [...]}`
- **Fallback:** `_review_spec_compliance_fallback()` if LLM response is unparseable
- **Pass** → node DONE with `{decision: pass, reason: ...}`
- **Fail** → node BLOCKED with reason. Executor retries with the specific error message injected.

#### BLOCKED vs FAILED distinction:

DONE means the node finished normally. BLOCKED means the Review Gate rejected the output but the node is still alive and can retry -- this is the explicit failure lane mechanism. The executor always knows exactly why it was rejected.

#### REVIEW\_SYSTEM prompt (common.py)

```
You are the Review Gate in a directed-graph agentic coding system (forge).
Review the generated files against the spec. Check:
1. All spec features are implemented
2. Code is syntactically correct
3. Files match the described file tree
4. Tests exist and cover the implementation
```

Respond ONLY with JSON: {"pass": true|false, "reason": "...",  
"issues": ["..."]}

## 4. Database -- 4-Tier Memory System (db.py)

---

Forge uses SQLite for all project memory. Each project gets its own database at `~/.forge/projects/<id>/memory.db`. The schema implements 4 tiers of memory with different retention characteristics.

### Schema

```
-- 4 memory tiers in one table, distinguished by tier column
CREATE TABLE memory (
 id INTEGER PRIMARY KEY,
 tier TEXT NOT NULL, -- short | mid | episodic | long
 agent TEXT, -- which agent wrote this
 key TEXT NOT NULL, -- namespace within tier
 value TEXT NOT NULL, -- JSON-serialized value
 created_at TEXT DEFAULT (datetime('now')),
 updated_at TEXT DEFAULT (datetime('now')),
 UNIQUE(tier, agent, key)
);

CREATE TABLE projects (
 id TEXT PRIMARY KEY,
 name TEXT NOT NULL,
 spec_md TEXT,
 created_at TEXT DEFAULT (datetime('now')),
 updated_at TEXT DEFAULT (datetime('now'))
);

-- Append-only SPEC.md changelog
CREATE TABLE spec_versions (
 version INTEGER PRIMARY KEY,
 prompt TEXT NOT NULL,
 spec_md TEXT NOT NULL,
 changelog TEXT NOT NULL,
 created_at TEXT DEFAULT (datetime('now'))
);

-- Subagent lifecycle tracking
CREATE TABLE subagent_runs (
 run_id TEXT PRIMARY KEY,
 task_id TEXT NOT NULL,
 agent_type TEXT NOT NULL,
 status TEXT NOT NULL, -- running | done | failed
 created_at TEXT DEFAULT (datetime('now')),
 finished_at TEXT,
 output_summary TEXT,
```

```
failure_reason TEXT
);
```

## The 4 Tiers

| Tier     | What it stores                                                                                   | Lifetime                        |
|----------|--------------------------------------------------------------------------------------------------|---------------------------------|
| short    | In-memory dict (STM). Scratchpad during a single node run.                                       | Cleared between runs            |
| mid      | Per-agent working memory. executor stores last_files_generated, spec_gen stores current_spec_md. | Survives retries within session |
| episodic | Full history of agent interactions: inputs, outputs, decisions.                                  | Pruned after 30 days            |
| long     | Cross-project knowledge. Manually curated patterns and facts.                                    | Permanent                       |

## Key DB Methods

- `write_memory(tier, agent, key, value)` -- upsert into memory table; JSON-serializes value
- `read_memory(tier, agent, key)` -- read one value; returns string or None
- `list_memory(tier, agent=None)` -- returns all keys in a tier
- `latest_spec_version()` → int -- highest version number from spec\_versions
- `get_spec_version(n)` → dict -- retrieves specific version with prompt/spec\_md/changelog
- `save_spec_version(version, prompt, spec_md, changelog)` -- append-only insert
- `create_subagent_run(run_id, task_id, agent_type)` -- inserts with status=running
- `finish_subagent_run(run_id, status, output_summary?, failure_reason?)` -- updates record
- `gc()` -- keeps last 10 spec\_versions, deletes episodic entries >30 days old

## 5. LLM Backends and Tool Calling (llm/backends/)

---

LLM backends are in `src/forge/llm/backends/`. All backends inherit from `LLMBackend` (**base.py**) which defines three abstract methods. A factory function `create_backend()` in `llm/__init__.py` instantiates the right backend from `LLMConfig`.

### LLMBackend Interface (base.py)

```
class LLMBackend(ABC):

 @abstractmethod
 def complete(self, prompt, system=None, *, max_tokens=4096,
 temperature=1.0, tools=None, tool_choice=None,
 **kwargs):
 """Synchronous completion. Returns LLMResponse."""
 ...

 @abstractmethod
 def complete_streaming(self, prompt, system=None, *,
 max_tokens=4096,
 temperature=1.0, tools=None, **kwargs):
 """Streaming. Yields content chunks as strings."""
 ...

 @abstractmethod
 def complete_json(self, prompt, system=None, *, max_tokens=4096,
 temperature=0.0, **kwargs):
 """Guaranteed JSON. Returns parsed dict."""
 ...
```

### Available Backends

| Backend                   | Provider  | Notes                                                                                                             |
|---------------------------|-----------|-------------------------------------------------------------------------------------------------------------------|
| <code>openai_.py</code>   | OpenAI    | GPT-4o / GPT-4-turbo. Official openai SDK. Tool calling via <code>function_call</code> parameter.                 |
| <code>anthropic.py</code> | Anthropic | Claude 3/4. Official anthropic SDK. Tool use via <code>tools[]</code> param, manual XML parsing for tool results. |

---

|                                |                |                                                                                                                                                                  |
|--------------------------------|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>minimax_openai.py</code> | MiniMax        | OpenAI-compatible HTTP API. HTTP POST to MiniMax endpoint with OpenAI request schema. Handles SSE streaming.                                                     |
| <code>mmx.py</code>            | MiniMax<br>CLI | MiniMax mmx CLI binary. Subprocess call to <code>mmx text chat</code> . Auth: <code>MINIMAX_API_KEY</code> env or <code>~/.mmx/config.json api_key</code> field. |
| <code>ollama.py</code>         | Ollama         | Ollama local models. Connects to <code>http://localhost:11434/v1/chat</code> . OpenAI-compatible API.                                                            |

## LLMConfig (llm/config.py)

`LLMConfig` is a Pydantic model holding `provider`, `model`, `api_key`, `base_url`, `max_tokens`, and `timeout`. It supports environment variable expansion (e.g. `${OPENAI_API_KEY}`). Loaded from `~/.forge/config.yaml` or environment variables.

## Tool Calling

Tool calling is supported on OpenAI and Anthropic backends. The `tools` parameter accepts a list of tool definitions in the provider's native format. When tools are provided, `LLMResponse` has a `tool_call` attribute. Callers (especially orchestrator and executor) check for tool calls and handle them appropriately.

## Streaming -- complete\_streaming()

`complete_streaming()` is a generator that yields content chunks as they arrive from the provider. `minimax_openai` handles SSE (Server-Sent Events) parsing. The TUI uses this for real-time token streaming display in the ChatScreen.

## 6. The Product Compiler

### (product\_compiler/)

---

The Product Compiler is a separate, heavier pipeline for building complete applications from scratch. It goes far beyond SPEC.md -- extracting intent through a conversational interview, designing a database schema, generating a full test suite (all initially failing), making them green, and reviewing the result against formal rules.

#### Pipeline Stages (pipeline.py, ProductCompilerPipeline class)

| Stage  | Agent               | Output                                                                                |
|--------|---------------------|---------------------------------------------------------------------------------------|
| 1      | InterviewerAgent    | IntentDocument (entities, relationships, actions, failure_rules, screens, user_types) |
| 2      | FlowDesignerAgent   | UserFlowTree (all user paths as a tree of UserFlow nodes)                             |
| 3      | ContractWriterAgent | List[APIContract] (request/response, read/write entities, error cases)                |
| 4      | RuleCompilerAgent   | RuleSet (IF/THEN rules -- atomic units of behaviour)                                  |
| Gate 1 | Human               | Rule Set approval. auto_approve skips this.                                           |
| 5      | SchemaDesignerAgent | DatabaseSchema (tables, columns, FKs, indexes, raw SQL)                               |
| Gate 2 | Human               | DB schema approval. auto_approve skips this.                                          |
| 6      | TestWriterAgent     | FailingTestSuite (all tests initially red)                                            |
| 7      | CoderAgent          | Makes tests green. Optional E2BSandbox.                                               |
| 8      | IntegratorAgent     | Runnable project (entry point, requirements.txt, README)                              |
| 9      | ReviewerAgent       |                                                                                       |

ReviewResult (Bandit, Radon, pyright + rule compliance)

## Pipeline Events (run()) is a Python generator

`pipeline.run()` is a generator that yields dicts with 'type' and payload. Enables both real-time TUI display and headless/non-interactive use.

```
yield {'type': 'stage', 'payload': {'stage': 'interview',
'description': '...'}}
yield {'type': 'question', 'payload': {'question': '...',
'history_len': N}}
yield {'type': 'thinking', 'payload': {'agent': 'interviewer',
'stage': '...'}}
yield {'type': 'output', 'payload': {'stage': '...', 'summary':
'...'}}
yield {'type': 'gate', 'payload': {'gate': 1, 'name': 'Rule
Compilation Approval', ...}}
yield {'type': 'approved', 'payload': {'gate': 1, 'mode':
'auto'|'user', ...}}
yield {'type': 'awaiting_input', 'payload': {'prompt': 'Approve rule
set? [y/n]'}}
yield {'type': 'complete', 'payload': {'state': {...}}}
```

## Key Models (models.py)

All pipeline stages communicate via typed dataclasses. No unstructured data passes between stages. Every field is typed, every optional field is explicit. All outputs are serializable to JSON for caching and audit.

| Model            | Purpose                                                                                                       |
|------------------|---------------------------------------------------------------------------------------------------------------|
| IntentDocument   | Complete product specification from interview. Entities, relationships, actions, failure_rules, user_screens. |
| UserFlowTree     | All user paths through the product. List of UserFlow nodes with trigger/conditions/outcome.                   |
| APIContract      | One user action mapped to HTTP request/response + side effects + error cases.                                 |
| RuleSet          | IF/THEN rules. Has approved_at/approved_by. Immutable once approved at Gate 1.                                |
| FailingTestSuite | All test cases (one per rule). Language/framework (python/pytest). All status=failing.                        |
| DatabaseSchema   | Tables, columns, FKs, indexes, raw SQL. Atlas migration_dir support.                                          |
| ReviewResult     | Static analysis + per-rule evidence. passed/failed + rules_violated list.                                     |

|              |                                                               |
|--------------|---------------------------------------------------------------|
| ChangeIntent | Edit mode: what is changing, what is preserved, blast radius. |
| DeltaRuleSet | Edit mode: preserve/change/new/remove rule diffs.             |

# 7. TUI -- Textual Full-Screen Interface (tui/)

`forge tui` opens a full-screen terminal UI built with Textual. It is the primary interactive experience, featuring streaming output, a model picker, and a command palette.

## App Structure (app.py, ForgeTUI class)

```
class ForgeTUI(App):
 BINDINGS = [
 ("ctrl+a", "open_model_picker", "Model"),
 ("ctrl+p", "open_command_palette", "Commands"),
 ("ctrl+c", "cancel_run", "Cancel"),
 ("escape", "go_home", "Home"),
]
```

## Screens

| Screen                | File                        | Purpose                                                             |
|-----------------------|-----------------------------|---------------------------------------------------------------------|
| HomeScreen            | screens/home.py             | Session browser / recent projects. Default landing screen.          |
| ChatScreen            | screens/chat.py             | Main chat + streaming output. Primary interaction screen.           |
| ModelPicker           | screens/models.py           | DataTable of available models. Ctrl+A opens it.                     |
| CommandPaletteDialog  | components/dialog.py        | Fuzzy command search. Ctrl+P opens it.                              |
| ProductCompilerScreen | screens/product_compiler.py | Interactive Product Compiler with real-time pipeline event display. |

## ChatScreen Layout (screens/chat.py)

Layout (top to bottom): Static (header-project), Static (header-model), Log (chat-log), StreamingOutput (streaming-output), Input (prompt-input).

### Flow when user submits a prompt

- `on_input_submitted()` captures value, clears input, appends to chat log as cyan "> prompt"
- posts `UserSubmitted(prompt)` message
- disables Input widget while streaming
- `on_streaming_output_streaming_finished()` re-enables input, refocuses
- Escape key: `action_cancel()` calls `finish_streaming()`

## Command Palette Commands

- `new chat --` opens new ChatScreen
- `forge new --` opens chat directly
- `forge compile --` opens ProductCompilerScreen
- `forge setup --` notifies user to run from terminal
- `exit --` closes TUI

## 8. Skills System (skills.py)

Skills are reusable procedural knowledge modules. Discovered from `~/.forge/skills/` and `~/.hermes/skills/` at startup. Each skill is a directory containing `SKILL.md` with YAML frontmatter.

### Skill Format

```

name: tdd
description: Test-driven development workflow
category: code

TDD Skill

Steps:
1. Write a failing test first
2. Run it to confirm it fails
3. Write minimal implementation
4. Refactor
```

### SkillRegistry Class

- `__init__()`: sets `search_paths` to `[.forge/skills, .hermes/skills]`, calls `_scan()`
- `_scan()`: recursively finds all `SKILL.md` files using `base.rglob('SKILL.md')`
- `_load_skill(path)`: parses YAML frontmatter (name, description, category), rest is body
- `match(task_type)`: exact name match first, then category prefix match. Returns list of matched Skill objects.
- `get(name)`: direct lookup by skill name
- `list_all()`: returns all registered skills

### Where Skills Are Used

| Component      | Task Type        | What is injected                   |
|----------------|------------------|------------------------------------|
| Spec Generator | "planning"       | Skill summaries into system prompt |
| Executor       | "code",<br>"tdd" | Skill summaries into system prompt |

|                 |            |                                          |
|-----------------|------------|------------------------------------------|
| Orchestrator    | agent_type | Skill content into delegation context    |
| SubagentManager | agent_type | Full SKILL.md content into system prompt |

---

## 9. MCP -- Model Context Protocol (mcp.py)

---

MCP integration lets Forge use external tools from MCP servers. Servers are defined in `~/.forge/mcp_servers.yaml`. Supports both stdio and HTTP transports. Currently implements JSON-RPC 2.0 over stdio.

### MCPConfig (loads from `~/.forge/mcp_servers.yaml`)

```
mcp_servers:
 filesystem:
 transport: stdio
 command: npx
 args: ["-y", "@modelcontextprotocol/server-filesystem", "/path"]
 env: {}
 slack:
 transport: http
 base_url: http://localhost:3000
 command: slack-mcp
 args: []
 env:
 SLACK_TOKEN: "${SLACK_TOKEN}" # env var expansion supported
```

### MCPClient Architecture (JSON-RPC 2.0 over stdio)

- `connect()`: starts MCP server subprocess, sends initialize request, lists available tools
- `_read_stdout()`: background thread reads stdout continuously, splits on newlines for JSON-RPC messages
- `_pending`: dict maps request IDs to (threading.Event, cached\_result) tuples
- `_send_request_sync()`: sends JSON-RPC request, waits for event (30s timeout), returns result
- `_send_notification()`: fire-and-forget JSON-RPC (no response expected)
- `list_tools()` → `list[MCPTool]`: returns cached list of available tools
- `call_tool(name, arguments)` → `MCPToolResult`: calls tools/call endpoint
- `is_alive()` → `bool`: True if subprocess is still running
- `disconnect()`: terminates subprocess gracefully (5s timeout then kill)

## How Executor Uses MCP

`_build_mcp_context()` in `executor.py` iterates all connected MCP clients and their tools, formats each as:

```
- server_name/tool_name: tool description
```

This string is injected into the Executor system prompt so the LLM knows what tools are available and can request them in its response.

# 10. File Service and Git-Aware Patch System

## FileService (file\_service.py)

FileService wraps git operations for the executor. Requires workdir to be a git repository (verified via `git rev-parse --is-inside-work-tree`). Raises `GitNotAvailable` if not.

| Method                     | Returns                     | Notes                                                                   |
|----------------------------|-----------------------------|-------------------------------------------------------------------------|
| <code>read(file)</code>    | <code>FileInfo</code>       | Content + git diff against HEAD. Binary files base64-encoded.           |
| <code>status()</code>      | <code>list[FileInfo]</code> | All files differing from HEAD: modified, untracked, deleted.            |
| <code>search(query)</code> | <code>list[str]</code>      | Fuzzy file search via <code>git ls-files</code> with substring scoring. |
| <code>list(dir)</code>     | <code>list[FileInfo]</code> | List files in directory, excluding dotfiles.                            |

## Patch System (patch.py)

Forge uses a custom surgical patch format instead of full-file replacements. This preserves git history and causes minimal diff noise. The LLM is explicitly told to prefer 'patch' action over 'write'/'update' for existing files.

### Patch Format (V4A format)

```
*** Begin Patch
*** Update File: path/to/file.py
@@ context line that uniquely identifies location
-old line to remove
+new line to add
 context line (unchanged, for anchoring)
*** End Patch

For new files:
*** Begin Patch
*** Add File: new_file.py
+line 1
+line 2
*** End Patch
```

```
For deletions:
*** Begin Patch
*** Delete File: unused.py
*** End Patch
```

## parse\_patch() -- Parsing

- Strips heredoc wrapper with regex
- Splits on `***` (Add|Update|Delete) File: header
- For Update: parses @@ chunks, extracts -lines (old) and +lines (new)
- Returns dict with list of PatchHunk objects

## apply\_patch() -- Application

- `seek_sequence()`: 4-pass matching -- exact, rstrip, strip, normalized unicode (smart quotes replaced)
- `compute_replacements()`: finds all chunk locations, sorts descending
- `apply_replacements()`: applies changes from bottom to top (reverse order preserves line numbers)
- Returns: {added: [...], modified: [...], deleted: [...]}

# 11. Subagent Delegation

## (agents.py)

---

SubagentManager spawns lightweight worker processes that receive a task, spec context, relevant skills, and workdir. They run the LLM as a subprocess and return a JSON result. This enables parallel work on complex subtasks.

### SubagentManager Lifecycle

- `spawn(task, agent_type, task_id, context) → SubagentResult`
- Generates `run_id`: `sub_<12 hex chars>`
- `create_subagent_run()` in DB records `status=running`
- `_build_system_prompt()`: constructs full system prompt with skills, project spec, and instructions
- `_build_user_prompt()`: creates the task description
- `_run_agent_process()`: calls LLM via `create_backend()` and `load_config()`
- `_parse_output()`: tries `JSON.parse` on response, falls back to raw string
- `finish_subagent_run()`: updates DB with `status=done|failed`

### System Prompt Construction

```
You are a {agent_type} subagent in a directed-graph coding system
(forge).
Your role: {task}

Project directory: {workdir}

Project Spec
{spec_md}

Existing Files
{list of files}

Your Skill
{skill content from SKILL.md if matched}

Instructions
1. Read the spec carefully
2. Write tests FIRST (TDD) before implementation
3. Write only files described in the spec
4. Stay within the project directory
```

```
5. Output JSON result: {"action": "done"|"error", "files_created": [...], "summary": "..."}

```

## 12. CLI Commands (cli.py)

The CLI entry point uses `@click` decorators. All commands accept `--provider` and `--model` flags to select the LLM backend. Shared state (`llm_config`) is passed via `click.Context (ctx.obj)`.

| Command                                        | Description                                                                        |
|------------------------------------------------|------------------------------------------------------------------------------------|
| <code>forge run &lt;prompt&gt;</code>          | Run a one-shot prompt through the full pipeline (orchestrator → executor → review) |
| <code>forge tui</code>                         | Open the full-screen Textual TUI                                                   |
| <code>forge compile [--auto-approve]</code>    | Run Product Compiler headless. <code>--auto-approve</code> skips both human gates. |
| <code>forge new [--name]</code>                | Create a new project and open its session                                          |
| <code>forge project &lt;id&gt; [prompt]</code> | Resume an existing project by ID, optionally send a follow-up prompt               |
| <code>forge projects</code>                    | List all projects with their current status and last updated time                  |
| <code>forge sessions</code>                    | List all sessions for the current project                                          |
| <code>forge kill [--session &lt;id&gt;]</code> | Mark a session as killed (graceful shutdown)                                       |
| <code>forge log [--session &lt;id&gt;]</code>  | Display the run log for a session                                                  |
| <code>forge ls</code>                          | List files in the project workspace                                                |
| <code>forge cat &lt;path&gt;</code>            | Display file contents                                                              |
| <code>forge rm &lt;path&gt;</code>             | Remove a file from workspace                                                       |
| <code>forge spec [--version N]</code>          | Show current SPEC.md or a specific version N                                       |
| <code>forge skills [task_type]</code>          | List available skills, optionally filtered by task type                            |
| <code>forge setup [--providers]</code>         | Interactive first-run setup: API keys, provider config                             |
|                                                | List available models for a provider                                               |

```
forge models [--
provider <name>]
```

```
forge loglevel [LEVEL] Set structlog verbosity (DEBUG|INFO|WARNING|
 ERROR)
```

## 13. Project Structure

```
forge/
├─ README.md
├─ SPEC.md # Full project specification
├─ pyproject.toml # Python package config (uv/pip),
dependencies
├─ src/forge/
│ └─ __init__.py
│ └─ cli.py # Click CLI entry point + all 17 commands
│ └─ graph.py # ForgeGraph: node/edge storage +
traversal
│ └─ db.py # 4-tier SQLite memory + project registry
│ └─ llm.py # create_backend(), load_config() factory
│ └─ llm/
│ └─ __init__.py # Exports LLMBackend, LLMResponse,
create_backend
│ └─ config.py # LLMConfig (Pydantic model)
│ └─ backends/
│ └─ base.py # LLMBackend abstract class
│ └─ openai_.py # OpenAI GPT-4o / GPT-4-turbo
│ └─ anthropic.py # Anthropic Claude 3/4
│ └─ minimax_openai.py # MiniMax OpenAI-compatible HTTP
│ └─ mmx.py # MiniMax mmx CLI subprocess
│ └─ ollama.py # Ollama localhost
│ └─ runners/
│ └─ common.py # System prompts, write_files(),
run_tests()
│ └─ orchestrator.py # Orchestrator node runner
│ └─ spec_gen.py # Spec Generator node runner
│ └─ executor.py # Executor node runner (TDD)
│ └─ review_gate.py # Review Gate node runner
│ └─ tui/
│ └─ app.py # ForgeTUI App + bindings
│ └─ context.py # AppContext (session/model state)
│ └─ screens/
│ └─ home.py, chat.py, models.py
│ └─ product_compiler.py
│ └─ components/
│ └─ output.py # StreamingOutput widget
│ └─ dialog.py # CommandPaletteDialog
│ └─ skills.py # SkillRegistry + Skill dataclass
│ └─ mcp.py # MCPClient + MCPConfig (JSON-RPC 2.0
stdio)
│ └─ agents.py # SubagentManager + SubagentResult
│ └─ file_service.py # FileService (git-aware file operations)
│ └─ patch.py # parse_patch(), apply_patch(),
seek_sequence()
│ └─ permissions.py # check_permission(), PermissionRequired
│ └─ retry.py # RetryPolicy, with_retry decorator
│ └─ product_compiler/
│ └─ pipeline.py # ProductCompilerPipeline (9-stage
```

```

orchestrator)
| └─ models.py # All typed dataclasses
| └─ messaging.py # NATS / stdout pub-sub layer
| └─ sandbox.py # E2BSandbox
| └─ agents/ # 13 specialist agents
| └─ interviewer.py, flow_designer.py
| └─ contract_writer.py, rule_compiler.py
| └─ schema_designer.py, test_writer.py
| └─ coder.py, integrator.py, reviewer.py
| └─ impact_analyst.py, rule_extractor.py
| └─ delta_compiler.py, blast_checker.py,
test_delta_writer.py
└─ tests/
 └─ docs/plans/

~/.forge/ (runtime config, created on first run)
└─ config.yaml # LLM provider settings
└─ mcp_servers.yaml # MCP server definitions
└─ skills/ # User-defined skills
└─ workspace/<id>/ # Per-project generated files
 └─ projects/<id>/memory.db # Per-project SQLite database

```

# 14. Key Design Principles

---

## 1. Explicit Failure Lanes

Every node has a defined failure mode. The Review Gate returning `BLOCKED` (not `FAILED`) means the node is alive and can retry with the specific failure reason. This is the core innovation over simple linear pipelines -- the executor always knows exactly why it was rejected and can course-correct on retry.

## 2. Append-Only Spec Changelog

SPEC.md versions are never overwritten. `db.save_spec_version()` inserts a new row each time with the full spec content and a changelog string. You can retrieve any version, diff between versions, or roll back. This creates a full audit trail of how the specification evolved.

## 3. TDD-First Executor

The executor generates tests BEFORE implementation. All tests are initially failing (the codebase does not exist yet). This forces precise specification of behaviour, prevents the LLM from writing code that looks right without being verifiable, and makes the Review Gate's pass/fail decision objective (pytest exit code) rather than subjective.

## 4. 4-Tier Memory for Persistent Context

short: in-memory dict cleared each node run. mid: survives retries within a session. episodic: full decision history. long: cross-project knowledge. This separation prevents context overflow while maintaining project continuity across sessions and runs.

## 5. Git-Aware Surgical Edits

`FileService.status()` shows what changed since HEAD. Executor injects current content of changed files into the LLM prompt. The patch format uses context lines to anchor changes, enabling surgical edits that preserve git history instead of full-file overwrites.

## 6. Human Gates in Product Compiler

The Product Compiler stops at two human gates: (1) after Rule Compilation, (2) after Database Design. `auto_approve=True` skips these for CI/automation. The gates force explicit human sign-off on the formal specification before code is generated, preventing scope creep and ensuring the rules truly reflect the user's intent.

## 7. Product Compiler = Formal Methods Lite

IntentDocument → UserFlowTree → APIContract → RuleSet → FailingTestSuite → Green Code → Review. Every stage output is a typed dataclass. The pipeline treats software development as a formal transformation: at each stage the output is deterministic, serializable, and auditable. The Reviewer uses static analysis tools (Bandit security checks, Radon cyclomatic complexity, pyright type errors) to ground LLM judgment in measurable evidence rather than vibes.

---

Generated from line-by-line source code analysis of shaheennazir/forgo v0.6  
All references are from actual source files