Large Language Models in Finance

APPENDICES

A Practitioner's Reference Volume

Juan F. Imbet

Paris Dauphine – PSL University

Paris, 2026

Companion to the main text. Cross-references of the form "Appendix A" or "Chapter 5" point between this volume and the main book.

This volume collects the appendices of *Large Language Models in Finance*; it is intended to be read alongside the main text.

First edition, 2026.

Contents

APPENDIX A

CLAUDE CODE: A PRACTITIONER'S REFERENCE

Remark A.1 (Learning Objectives). After working through this appendix, the reader should be able to:

1. Install Claude Code, authenticate it, and configure a project-level `CLAUDE.md` and `settings.json` with appropriate permission and deny-list entries.
2. Describe the built-in tool set (Read, Write, Edit, Bash, Glob, Grep, Agent, Workflow, Skill) and explain which tool to use for a given task.
3. Write a `SKILL.md` that orchestrates two or more agents in sequence, and register it as a slash command.
4. Configure a `PostToolUse` hook that runs a linter after any file edit.
5. Design a Workflow script that fans work out across multiple agents in parallel and returns structured JSON results.
6. Identify the security risks relevant to deploying Claude Code on financial data and apply the deny-list and secrets-management mitigations.
7. Map at least three quantitative-finance research tasks to concrete Claude Code workflows involving skills, hooks, and scheduled routines.

A.1 What is Claude Code

THE BIGGER PICTURE

On 27 February 2025, Anthropic released a research preview of a terminal-native coding assistant named Claude Code. The reception was unusual: engineers who tried it reported that it was the first AI coding tool that could take a vague description of a task and actually complete it across dozens of files without constant hand-holding. Within thirteen months, 4% of all public GitHub commits—roughly 135,000 per day—were attributed to Claude Code, a 42,896× growth rate (Anthropic 2025c). By early 2026, Anthropic reported that 90% of

its own production code was AI-written, predominantly via Claude Code. The tool had graduated from curiosity to infrastructure.

A.1.1 Overview and Positioning

Claude Code is an *agentic* coding assistant that runs inside a terminal (or an IDE terminal panel) and has persistent access to the developer's file system, shell, and external services. Unlike chat-based assistants that operate purely in a conversation window, Claude Code can read files, write files, run shell commands, spawn background processes, call external APIs via Model-Context-Protocol servers, and coordinate fleets of parallel sub-agents.

AI coding assistants that generate code from natural language prompts have been studied since at least Chen et al. (2021), whose Codex model underpins GitHub Copilot. Controlled experiments subsequently showed productivity gains of roughly 55% for tasks assisted by such tools (Peng et al. 2023). What distinguishes Claude Code from those earlier systems is architectural: the model is placed *in the loop of the build process* rather than acting as a side-channel advisor. When the model writes a function, it can immediately compile it, run the test suite, read the error output, and iterate—all within a single conversation turn if the user's permission settings allow it.

A.1.2 Supported Models

Claude Code works with the full range of production Anthropic models. As of mid-2026, the three actively supported model tiers are:

- **Claude Opus 4.8** — the flagship reasoning model; required for Agent Teams (Section A.7.4) and recommended for complex multi-file refactors.
- **Claude Sonnet 4.6** — the default model; balances capability and cost for day-to-day coding tasks and lecture drafting.
- **Claude Haiku 4.5** — the fast, low-cost model; useful for high-frequency sub-agent calls (e.g., batch file classification) where latency and cost dominate accuracy concerns.

The active model can be overridden per session with the `/model` command or set permanently in `settings.json`. Individual sub-agents and Workflow stages can specify their own model via the `model` option, enabling cost-aware tiering within a single orchestration run.

A.1.3 Key Milestones

The evolution of Claude Code's feature set matters for practitioners because community resources and blog posts are often written against older versions. Table A.1 summarises the major capability releases.

Table A.1. Major Claude Code capability releases (2024–2026).

Date	Feature	Notes
November 2024	MCP support	Model Context Protocol; first external tool bridge
February 2025	Research preview	Public launch with Bash, Read, Edit, Write tools
July 2025	Subagents (Agent tool)	Spawning isolated child sessions
September 2025	Hooks	Pre/post tool-call shell scripts
October 2025	Skills & Plugins	Markdown slash commands; bundled plugin packag
October 2025	Workflow tool	Deterministic multi-agent fan-out scripts
February 2026	Agent Teams	Peer-to-peer multi-session coordination
March 2026	Scheduled Routines	Cron-style remote execution on Anthropic infra

A.2 Installation and Setup

A.2.1 Prerequisites and Authentication

Claude Code requires Node.js 18 or later. Installation is a single `npm` command:

```
npm install -g @anthropic-ai/claude-code
```

On first launch, Claude Code opens a browser-based OAuth flow to authenticate against an Anthropic account. For headless servers (CI/CD environments, remote compute nodes), set the `ANTHROPIC_API_KEY` environment variable instead; the browser flow is then skipped.

```
export ANTHROPIC_API_KEY="sk-ant-..."
claude                          # start interactive session
claude "fix all type errors"  # non-interactive single-shot mode
```

To run Claude Code with a specific model from the command line:

```
claude --model claude-opus-4-8
```

A.2.2 IDE Integration

Claude Code ships first-party IDE extensions for VS Code and JetBrains IDEs (IntelliJ IDEA, PyCharm, WebStorm, etc.). The extensions embed the full Claude Code terminal panel inside the IDE, with additional GUI affordances:

- **Inline diffs** — proposed file edits are shown as unified diffs inside the editor gutter before the user approves them.
- **Plan review pane** — in Plan mode (Section A.3.2), the plan document is rendered in a dedicated editor panel and can be edited directly.
- **`@`-mention** — any open file can be dragged into the conversation with `@filename` to provide immediate context.
- **Conversation history** — previous sessions are browsable via the IDE's activity bar.

Install the VS Code extension from the Marketplace by searching *Claude Code* (publisher: `Anthropic`). For the integrated terminal to support multi-line input via SHIFT+ENTER, run the following inside a Claude Code session once after installation:

```
/terminal-setup
```

This writes the required keybinding to VS Code's `keybindings.json`.

A.3 Core Interface

A.3.1 CLI Commands and Navigation

Claude Code's slash commands are the primary navigation layer. Table A.2 lists the most commonly used built-in commands.

Any skill installed in `.claude/skills/` is also accessible as a slash command (e.g., `/fetch-filings`, `/run-backtest`).

A.3.2 Plan Mode

Plan mode is a read-only exploration mode: Claude Code can analyse the codebase, reason about architecture, and produce a structured plan, but it cannot execute any writes or shell commands. Use Plan mode to understand what a large, ambiguous task will require before granting execution permissions.

Table A.2. Built-in Claude Code slash commands.

Command	Effect
`/help`	List all available commands and skills
`/model [name]`	Switch the active model
`/compact`	Compress conversation history to free context
`/clear`	Start a fresh session (context wiped)
`/config`	Open interactive settings editor
`/permissions`	Add or remove tool allowlist entries
`/memory`	View and edit auto-memory files
`/status`	Show current session token usage
`/fast`	Toggle Fast mode (Opus with faster output)
`/terminal-setup`	Write IDE keybindings for multi-line input
`/init`	Generate a `CLAUDE.md` for the current repo
`/workflows`	Show live progress of running Workflow tasks

Activate Plan mode by pressing SHIFT+TAB once from the default Edit mode; subsequent presses cycle through Auto-Accept mode and back. The three modes are:

1. **Edit mode** (default) — Claude proposes each file change individually; the user approves or rejects before writing.
2. **Plan mode** — read-only; Claude explores and writes a plan document; no files are modified.
3. **Auto-Accept mode** — Claude executes approved tool categories without per-call confirmation; see Section A.9.1 for how to configure which tools are auto-approved.

UNDER THE HOOD

In Plan mode, Claude Code internally activates the `EnterPlanMode` and `ExitPlanMode` tools. These tools are deferred-schema tools: their parameter schemas are not loaded until explicitly requested, which is why Plan mode imposes essentially zero overhead on sessions that never use it. This pattern—load schemas on demand—is used broadly in Claude Code to keep the system prompt size bounded even when hundreds of extension tools are registered.

A.3.3 Keyboard Shortcuts

The most productive keyboard shortcuts differ slightly between macOS and Windows/Linux. Table A.3 gives the canonical bindings.

Table A.3. Claude Code keyboard shortcuts (macOS / Windows–Linux).

Action	macOS	Windows/Linux
Submit prompt (single line)	RETURN	RETURN
Newline in prompt	SHIFT+RETURN	SHIFT+RETURN
Cycle Edit → Plan → Auto	SHIFT+TAB	SHIFT+TAB
Open plan file in editor	CMD+G	CTRL+G
Undo last AI change	ESC, ESC	ESC, ESC
Toggle extended thinking	OPTION+T	ALT+T
Toggle editor/Claude focus	CMD+ESC	CTRL+ESC
Interrupt running agent	CTRL+C	CTRL+C

The double-ESC rewind is particularly useful during iterative drafting: it undoes all file changes from the last assistant turn and removes that turn from the conversation history, allowing a clean retry with a different prompt.

A.3.4 Context Compaction and Conversation Management

Claude Code's context window grows with every tool call and assistant response. When the context approaches its limit, Claude Code automatically runs `/compact`: it summarises the conversation history into a compressed representation that preserves task state while freeing tokens for continued work.

Key points for practitioners:

- Running `/compact` *manually* before starting a new phase of work gives more predictable results than waiting for the automatic trigger.
- The Anthropic prompt cache has a 5-minute TTL. Sleeping longer than 300 seconds between turns (e.g., during a manual review step) incurs a cache miss; in the Workflow tool the scheduler accounts for this when choosing polling intervals.
- For very long autonomous runs, consider splitting the task into independent Workflow phases; each phase starts with a fresh agent and a focused prompt rather than a compacted summary of a long history.

A.4 Configuration and Memory

A.4.1 The CLAUDE.md File Hierarchy

`CLAUDE.md` is the primary instruction file: Claude Code reads it automatically at the start of every session and treats its content as highest-priority guidance, overriding

default system-prompt behaviour. The file system supports a layered hierarchy:

1. **Global** (`~/.claude/CLAUDE.md`) — applies to all projects on the machine; a good place for personal preferences and cross-project conventions.
2. **Project** (`.claude/CLAUDE.md` at the repo root) — applies to the current project; checked into version control so all collaborators (human and AI) see the same rules.
3. **Sub-directory** (any `CLAUDE.md` in a nested folder) — automatically included when Claude Code works within that directory; useful for monorepos with per-package conventions.

Claude Code loads and merges all three levels, giving the more specific file precedence on any conflict. The `/init` command generates a starter `CLAUDE.md` by reading the current directory structure, recent git history, and detected languages/frameworks.

UNDER THE HOOD

`CLAUDE.md` is not limited to free-form prose. It can contain structured YAML front matter (useful for storing project-level configuration such as data paths, model parameters, or quality thresholds), conditional blocks, and even embedded skill references. Claude Code does not execute front matter directly; rather, skills and agents read it programmatically using the `Read` tool to extract configuration. This indirection—human-readable markdown that is also machine-parseable—is what allows a single configuration file to govern both AI sessions and the shell hooks that validate them.

A.4.2 Settings, Permissions, and Environment Variables

Three JSON settings files govern runtime behaviour, one at each scope:

```
// ~/.claude/settings.json       (global - lowest priority)
// .claude/settings.json         (project - medium priority)
// .claude/settings.local.json   (session - highest priority; gitignored)
```

The most important keys are:

```
{
  "model": "claude-sonnet-4-6",
  "permissions": {
```

```
    "allow": [
      "Bash(git log:*)",
      "Bash(npm test:*)",
      "Edit",
      "Read"
    ],
    "deny": [
      "Bash(rm -rf:*)",
      "Bash(curl:*)"
    ]
  },
  "env": {
    "ANTHROPIC_API_KEY": "${ANTHROPIC_API_KEY}",
    "PYTHONPATH": "${workspaceRoot}/src"
  },
  "hooks": {
    "PostToolUse": [
      { "matcher": "Edit", "command": "bash .claude/hooks/lint.sh" }
    ]
  }
}
```

The `permissions.allow` list uses `Tool(pattern:*)` syntax; the pattern matches against the tool's first argument. The `permissions.deny` list takes precedence over `allow`: if a command matches both, Claude Code denies it. This asymmetric design ensures that safety constraints cannot be accidentally overridden by a broad allowlist entry.

The `/permissions` slash command provides an interactive editor for the allowlist that does not require manually editing JSON.

Environment variables. Variables declared in `settings.json` under `env` are injected into every Bash tool call. Use `${VAR}` syntax to forward a variable from the parent shell. Never hardcode API keys in settings files that are committed to version control; use `settings.local.json` (gitignored by default) or OS-level secret stores instead.

A.4.3 Auto-Memory System

Claude Code maintains a persistent memory store at `~/.claude/projects/<encoded-path>/memory/`. Each memory is a markdown file with YAML front matter:

```
---
name: project-data-conventions
description: Return convention for this project is log returns, not simple
metadata:
```

```
  type: project
---

Always compute log returns: `np.log(price / price.shift(1))`.
**Why:** Risk models and factor regressions assume additive returns; simple
returns compound multiplicatively and bias long-horizon estimates.
**How to apply:** Flag any code that uses `price.pct_change()` without a
subsequent log transformation.
```

The memory system has four types: **user** (the human's role and preferences), **feedback** (corrections and confirmations), **project** (goals, deadlines, decisions), and **reference** (pointers to external systems). A `MEMORY.md` index file is always loaded into context; the full memory files are read on demand when flagged as relevant by the index.

Use the `/memory` command to inspect and edit the memory store from within a Claude Code session.

A.5 Built-in Tools

Claude Code exposes a curated set of tools to the language model. Each tool call requires user approval unless the tool (or a matching pattern) is listed in `permissions.allow`.

A.5.1 File System Tools

`Read` Reads a file at an absolute path. Supports line-range offsets (`offset`, `limit`) for large files; for PDFs accepts a `pages` range. Always read a file before editing it: the `Edit` tool refuses to operate on files it has not seen.

`Write` Overwrites a file with new content. Use only for new files or complete rewrites; prefer `Edit` for partial changes.

`Edit` Applies an exact string replacement (`old_string` → `new_string`) in a file. The `replace_all` flag performs the substitution everywhere in the file, useful for renaming a symbol throughout a module. The replacement fails if `old_string` is not unique, preventing accidental edits.

`Glob` Fast file-pattern matching. Returns paths sorted by modification time. Use instead of `find` for any file-discovery task.

`Grep` Content search backed by `ripgrep`. Supports full regex syntax, file-type filtering (`type` parameter), multi-line matching, and context lines (`-C`). Output modes: `files_with_matches`, `content`, or `count`.

A.5.2 The Bash Tool

The `Bash` tool executes arbitrary shell commands with a configurable timeout (up to 600 s). Working directory persists across calls within a session; environment state (variables, functions) does not.

Commands can be run in the background by setting `run_in_background: true`, in which case Claude Code is notified when the process completes rather than blocking. This is essential for long compilation jobs or test suites.

```
# Example: run a backtest in the background
python scripts/run_backtest.py --strategy momentum --start 2015-01-01
```

Security note. The Bash tool is the highest-risk tool because it can execute arbitrary code. Use the `permissions.deny` list to block patterns that should never run in your environment (e.g., `curl` calls to external hosts, `rm -rf` without path specifics).

A.5.3 Orchestration Tools

Three tools control multi-agent execution:

`Agent` Spawns a child Claude Code session with its own context window, prompt, and tool set. Returns the agent's final text output (or a structured JSON object if a `schema` is provided). Useful for parallelising independent tasks or protecting the main context window from excessive search results. The optional `isolation: "worktree"` flag runs the agent in a fresh git worktree, preventing file conflicts when multiple agents write in parallel.

`Skill` Invokes a named skill (slash command) from `.claude/skills/` and returns its output. Equivalent to typing `/skill-name` at the prompt, but callable programmatically from within another skill or Workflow script.

`Workflow` Executes a JavaScript orchestration script that fans out work across many agents deterministically. See Section A.7.3 for the full API.

A.5.4 Extended Thinking

Claude Opus models support *extended thinking*: an internal chain-of-thought scratch-pad in which the model reasons before producing output. Claude Code exposes this via the `Alt+T` / `Option+T` keyboard shortcut or the `/fast` toggle (which disables thinking in exchange for lower latency).

Extended thinking is most valuable for:

- architectural decisions that span many files,
- mathematical derivations that require multi-step symbolic reasoning, and
- debugging sessions where the root cause is non-obvious.

When extended thinking is active, the model's reasoning tokens are *not* billed at the same rate as output tokens; consult the Anthropic pricing page for current rates.

A.6 Extension Mechanisms

A.6.1 Skills

A *skill* is a markdown file at `.claude/skills/<name>/SKILL.md` that describes a multi-step workflow. When the user types `/<name>` at the prompt, Claude Code loads the file and follows its instructions exactly. Skills are:

- **Portable** — the same `SKILL.md` file works in Claude Code, Codex CLI, Gemini CLI, and Cursor without modification.
- **Version-controlled** — stored in the repo alongside the code they govern; changes are visible in `git log`.
- **Composable** — a skill can invoke other skills via the `Skill` tool, enabling pipelines like `/fetch-filings` → `/extract-sentiment` → `/build-factors`.

Skills live in `.claude/skills/`; run `/help` from any session to list those available in the current project.

A.6.2 Hooks

Hooks are shell scripts registered in `settings.json` that execute automatically in response to tool-use events. The available hook events are:

`PreToolUse` Fires before a tool call; can cancel the call by exiting non-zero.

`PostToolUse` Fires after a tool call completes; its output is shown to the model as feedback.

`UserPromptSubmit` Fires when the user submits a prompt; can prepend additional context.

`Stop` Fires when Claude Code finishes a turn; useful for notifications or cleanup.

```
1 "hooks": {
```

```
  "PostToolUse": [
    {
      "matcher": "Edit",
      "command": "bash .claude/hooks/lint-on-save.sh \"$CLAUDE_TOOL_INPUT_FILE\""
    }
  ],
  "Stop": [
    { "command": "bash .claude/hooks/notify.sh" }
  ]
}
```

Hook scripts receive tool input and output as environment variables and can write to `stdout` to inject messages into the conversation. A common pattern for quantitative projects is a `PostToolUse` hook on `Edit` that re-runs a data-validation script whenever a model configuration file changes, surfacing schema errors before the next agent step.

A.6.3 MCP Server Integration

The Model Context Protocol (MCP) is an open standard (released November 2024) (Anthropic 2024) that allows Claude Code to connect to external processes that expose tools and data sources. An MCP server is a long-running process that Claude Code discovers via `settings.json`:

```
{
  "mcpServers": {
    "context7": {
      "command": "npx",
      "args": ["-y", "@upstash/context7-mcp@latest"]
    },
    "sqlite": {
      "command": "uvx",
      "args": ["mcp-server-sqlite", "--db-path", "data/research.db"]
    }
  }
}
```

MCP servers expose their tools as callable functions inside Claude Code's tool list. The finance-relevant servers include database connectors (SQLite, PostgreSQL), browser automation (for scraping financial filings), internal API bridges (Bloomberg terminal, FactSet), and document stores (Google Drive, email).

Tools from MCP servers are deferred: their parameter schemas are not loaded into context until the user invokes a `ToolSearch` query, keeping the system prompt compact when many servers are registered.

A.6.4 Plugins

A *plugin* (introduced October 2025) is a packaged bundle that can include any combination of skills, MCP server configurations, sub-agent definitions, and hooks. Plugins are installed with:

```
claude plugin install @anthropic/financial-data-plugin
```

After installation, all of the plugin's components are available without manual configuration. The community plugin registry hosts hundreds of domain-specific plugins; for regulated environments, private registries can be pointed to via the `pluginRegistry` setting.

A.7 Agents and Multi-Agent Systems

THE BIGGER PICTURE

The move from a single LLM call to a *system* of coordinated LLM calls is arguably the most consequential architectural shift in applied AI since the Transformer itself. Early demonstrations used independent agents with shared memory to simulate complex social dynamics (Park et al. 2023); more recent work has evaluated LLM systems against real-world software engineering tasks at scale (Jimenez et al. 2024). A single model, no matter how capable, operates within a fixed context window; a system of models can divide a large task into pieces, process them in parallel, check each other's work, and synthesise results that no single context window could hold. Claude Code's multi-agent architecture is designed around exactly this observation.

A.7.1 Subagents and the Agent Tool

The `Agent` tool spawns a child Claude Code session. Each child has:

- its own independent context window,
- a prompt written by the parent agent,
- a configurable tool set (defaulting to all tools), and
- an optional `schema` that forces the child to return a validated JSON object rather than free text.

Children run concurrently when multiple `Agent` calls appear in the same response turn. The parent collects results and synthesises them. The most common pattern is a two-level hierarchy: a *planner* agent that breaks the task into pieces, followed by a fleet of *worker* agents that execute each piece in parallel.

```
// Simplified Agent call (as JSON schema)
{
  "description": "Classify sentiment of an earnings call transcript",
  "prompt": "Read data/transcripts/AAPL_2024Q4.txt and classify overall ...",
  "schema": {
    "type": "object",
    "properties": {
      "sentiment": { "type": "string", "enum": ["positive","neutral","negative"] },
      "confidence": { "type": "number" }
    }
  }
}
```

A.7.2 Custom Subagent Types

Named sub-agent types are defined as markdown files in `.claude/agents/`. Each file specifies a persona, a set of inputs, a task description, and constraints. When an `Agent` call includes `agentType: "risk-checker"`, Claude Code reads `.claude/agents/risk-checker.md` and prepends it to the child's system prompt.

A finance research project might define agent types such as:

- `data-validator` — checks that a dataset conforms to expected schema, return conventions, and date ranges; returns `PASS` or `FAIL` with a list of issues.
- `factor-builder` — constructs Fama–French-style factor portfolios from cleaned price and fundamentals data.
- `risk-checker` — verifies that a proposed portfolio does not breach position limits, sector concentration caps, or drawdown thresholds.
- `report-writer` — drafts regulatory or client-facing reports from structured backtest output, following a house style guide.
- `literature-reviewer` — scans SSRN and arXiv for recent papers on a given topic and returns structured BibTeX entries.

A.7.3 The Workflow Tool

The `Workflow` tool runs a JavaScript orchestration script that fans work out across many agents deterministically. Unlike the `Agent` tool (which is invoked by the model

and subject to the model's discretion), a Workflow script is controlled by explicit JavaScript control flow—loops, conditionals, and fan-out—making the execution pattern reproducible and auditable.

A minimal Workflow script that classifies the sentiment of multiple earnings call transcripts concurrently illustrates the core pattern:

```
export const meta = {
  name: 'classify-earnings-calls',
  description: 'Sentiment-classify earnings transcripts in parallel',
  phases: [{ title: 'Classify' }],  // progress group shown in /workflows
}

const tickers = ['AAPL', 'MSFT', 'JPM', 'GS', 'BAC']

// JSON Schema forces each agent to return structured data, not free text.
const SENTIMENT_SCHEMA = {
  type: 'object',
  properties: {
    ticker:     { type: 'string' },
    sentiment:  { type: 'string', enum: ['positive', 'neutral', 'negative'] },
    confidence: { type: 'number' },
    key_phrases: { type: 'array', items: { type: 'string' } },
  },
}

// parallel() fans out: all five agents start immediately and run concurrently.
// Wall-clock time = slowest single agent, not sum of all five.
const results = await parallel(tickers.map(t => () =>
  agent(
    `Read data/transcripts/${t}_2024Q4.txt. Classify sentiment and extract
     up to five key forward-looking phrases.`,
    { label: `classify:${t}`, phase: 'Classify', schema: SENTIMENT_SCHEMA }
  )
))

// results is an array of validated objects: [{ ticker: 'AAPL', sentiment:
↪  'positive', ... }, ...]
return results
```

Running this script with `/workflows` open shows five concurrent "classify:TICKER" agents in the *Classify* group. Each returns a validated JSON object matching `SENTIMENT_SCHEMA`; if a model response does not conform to the schema, the runtime retries automatically before failing.

Key Workflow primitives:

`agent(prompt, opts)` Spawns one sub-agent; returns the agent's text output or,

when `opts.schema` is provided, a validated JSON object.

`parallel(thunks)` Runs all thunk functions concurrently; waits for all to complete (barrier); returns an array of results.

`pipeline(items, ...stages)` Runs each item through all stages without inter-stage barriers. The default choice: use `parallel` only when all prior-stage results are genuinely needed before the next stage can start.

`phase(title)` Opens a named progress group; subsequent `agent` calls are grouped under this title in the live `/workflows` view.

`log(message)` Emits a progress message visible in the UI.

Workflows support *resume*: if a run is interrupted, re-launching with the same `scriptPath` and `resumeFromRunId` replays cached agent results and continues from the first changed or new call.

A.7.4 Agent Teams

Agent Teams (released February 2026) extend the parent/child subagent model to a *peer* coordination model. A team consists of:

Team Lead The main Claude Code session that initialises the team, posts the initial task list, and synthesises final results.

Teammates Independent Claude Code processes, each with their own context window. Teammates can read from and write to a shared task list, and can send direct messages to each other via a mailbox system.

Agent Teams require Claude Opus 4.8 for all participants and are disabled by default. Enable them by setting the environment variable `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1`. They are most productive for tasks that benefit from genuine collaboration: parallel code review, complex debugging where teammates test competing hypotheses, or large feature implementations where different teammates own different layers (frontend, backend, database, tests).

Token consumption is roughly 7× a single-session Plan-mode run for comparable tasks; reserve Agent Teams for problems that would otherwise require repeated manual context-switching.

A.8 Automation and Remote Execution

A.8.1 Scheduled Routines

As of March 2026, Claude Code supports *Routines*: cron-style tasks that execute on Anthropic-managed infrastructure, decoupled from the developer's local machine. Create a routine with the `/schedule` command:

```
/schedule "every weekday 06:00 UTC: /fetch-factor-returns, validate schema, commit"
```

Routines are stored in `.claude/routines/` and have the same access to project skills and MCP servers as an interactive session. Common use cases for quantitative finance workflows:

- Nightly factor return downloads from WRDS or Ken French's data library, with automatic schema validation before appending to the master dataset.
- Weekly model recalibration reports that re-estimate factor loadings, flag any Sharpe ratio regression, and post a summary to Slack.
- Post-merge checks that re-run all backtest notebooks and compare performance metrics against a stored baseline, alerting on regressions.

A.8.2 External Integrations

Claude Code integrates with Slack as a first-party surface. After authorising the integration, mentioning `@Claude` in a Slack channel or thread triggers a Claude Code session on Anthropic infrastructure; the session has access to the linked repository and can commit results back to GitHub.

For CI/CD integration, Claude Code can be invoked non-interactively:

```
# In a GitHub Actions workflow step:
claude --non-interactive \
       --model claude-sonnet-4-6 \
       "Run the backtest suite on all modified strategy files in this PR. \
        Fail if any strategy's Sharpe ratio drops below 0.5."
```

The `-non-interactive` flag disables the permission-approval UI; the allowlist in `.claude/settings.json` governs what the agent is permitted to do.

A.9 Security, Privacy, and Cost

A.9.1 The Permissions Model

Claude Code's permission system follows a *default-cautious* principle: every tool call that is not explicitly allowed prompts the user for approval. The layered allowlist/denylist design described in Section A.4.2 provides fine-grained control.

For shared codebases, commit a minimal `.claude/settings.json` that allows only the operations required for automated tasks (read-only tools plus specific git operations) and requires human approval for anything that touches production data or external network endpoints.

The `permissions.deny` key provides a hard veto. Any pattern listed there is blocked regardless of what the allowlist says or what the model requests. Finance teams should at minimum deny:

```
"deny": [
  "Bash(curl:*)",
  "Bash(wget:*)",
  "Bash(pip install:*)",
  "Bash(npm install:*)",
  "Bash(rm -rf:*)"
]
```

This prevents the model from inadvertently (or through a prompt-injection attack) exfiltrating data or installing arbitrary packages.

A.9.2 Sensitive Data and Secrets Management

Claude Code reads the project directory, which may contain data files, model weights, or configuration with credentials. Best practices:

1. **Gitignore secrets** — add `.env`, `*.pem`, and `secrets.json` to `.gitignore` before running `/init`, which reads git history.
2. **Deny read access** — `"deny": ["Read(.env*)", "Read(*credentials*)"]` prevents the model from inadvertently including secret values in its context.
3. **Use `settings.local.json`** — environment variables injected via this file are not committed to the repository; use it for per-developer API keys.
4. **Redact before sharing transcripts** — the `/export` command saves the conversation; review the export before sharing, as it may capture tool outputs that include file contents.

For financial data subject to NDA or data sharing agreements, consider running Claude Code with a local model endpoint (via the `baseURL` setting) rather than the Anthropic API, so data never leaves your perimeter.

A.9.3 Token Consumption and Cost Control

Claude Code sessions can consume substantial tokens, particularly when extended thinking is active or when Workflow scripts run large agent fleets. Table A.4 gives approximate consumption for representative finance research tasks.

Table A.4. Approximate token consumption for representative Claude Code tasks (Sonnet 4.6).

Task	Input tokens	Output tokens
Classify one earnings call transcript	10,000–20,000	1,000–2,000
Parallel sentiment: 10 transcripts	100,000–200,000	10,000–20,000
Full codebase review (multi-agent)	80,000–200,000	15,000–40,000
Agent Teams run (7 sessions)	350,000+	70,000+

Cost-reduction strategies:

- Use Sonnet 4.6 for bulk operations (scoring, proofreading) and reserve Opus 4.8 for tasks requiring deep reasoning.
- Exploit prompt caching: CLAUDE.md and long system prompts are cached by Anthropic with a 5-minute TTL; rapid iteration within a session benefits automatically.
- Set the `budget` directive in Workflow scripts (`+500k tokens`) to cap a run and prevent runaway agent loops.
- Run `/compact` at natural task boundaries rather than letting the context grow until automatic compaction triggers.

A.10 Claude Code in Research and Finance Workflows

A.10.1 Use Cases for Quants and Researchers

The following use cases have proven especially productive for quantitative finance researchers and practitioners:

Literature synthesis A Workflow script that fans out across a list of SSRN or arXiv paper IDs, has one sub-agent per paper extract methodology and key results as

structured JSON, then has a synthesis agent produce a comparative table—in minutes rather than days.

Code archaeology Dropping a legacy pricing library into the project directory and asking Claude Code to generate a call graph, identify undocumented assumptions, and produce a test suite covering edge cases.

Report drafting Using a `report-writer` agent iteratively to produce investment memos or regulatory reports that meet a house style and a readability threshold, with a `PostToolUse` hook that checks compliance terminology on every edit.

Data pipeline debugging Providing a failing notebook and asking Claude Code to reproduce the error, hypothesise causes, write a minimal reproducer, and propose a fix—all within one non-interactive `claude` invocation.

Nightly model monitoring A scheduled routine that runs a battery of sanity checks on a live NLP model (distribution shift, calibration, new OOV tokens), commits a JSON report, and posts a summary to Slack if any check fails.

Example A.2 (End-to-end: building an earnings-call sentiment factor). Suppose a quant researcher wants to build a long/short equity factor based on earnings-call sentiment for S&P 500 firms. The following session illustrates a complete Claude Code workflow from raw transcripts to a factor return series.

```
# Step 1: Use Plan mode to review the task scope before touching any files
# (Press Shift+Tab to enter Plan mode, then submit the prompt)
"Outline a pipeline to classify Q4 2024 earnings calls for S&P 500 firms,
 construct a long-short sentiment factor, and backtest it 2015-2024."

# Step 2: Approve the plan, switch back to Edit mode, then run the fetch skill
/fetch-filings --source seekingalpha --quarter 2024Q4 --universe sp500

# Step 3: Fan out sentiment classification across all transcripts (Workflow)
/classify-sentiment --input data/transcripts/2024Q4/ --output data/sentiment/

# Step 4: Build the factor and run the backtest
claude "Using data/sentiment/2024Q4_scores.csv, construct a monthly
        long-short portfolio: long top-quintile sentiment, short bottom-quintile.
        Use equal weights within each leg. Compute net returns 2015-2024."

# Step 5: Validate and commit
/validate-backtest results/sentiment_factor_2015_2024.csv
git add data/sentiment/ results/ && git commit -m "feat: Q4-2024 sentiment factor"
```

At Step 3, the `/classify-sentiment` skill launches a Workflow that runs one subagent per transcript in parallel; wall-clock time for 500 transcripts is roughly equal to the slowest single classification rather than the sum. The researcher reviews the plan at each stage and uses the double-Esc rewind to undo any step that produces unexpected output.

A.10.2 Recommended Workflow Patterns

1. **Plan before writing.** Use Plan mode to review the proposed changes before committing to Auto-Accept. For any new pipeline or feature, enter Plan mode first to inspect the intended file changes, then switch to Edit mode to execute.
2. **Validate early, validate often.** Run your validation skill after every significant change, not just at the end. Structured JSON output from a validator agent provides a more reliable compass for the next iteration than subjective impression.
3. **Separate concerns with agents.** Do not ask a single session to draft content, check math, and improve style in the same turn. Each task is well-suited to a different agent persona; the `Agent` tool makes this separation cheap.
4. **Use the deny list proactively.** In research environments with sensitive data, configure `permissions.deny` before the first session, not after a concern arises.
5. **Compact at phase boundaries.** Run `/compact` when transitioning from drafting to reviewing, or from reviewing to releasing. This preserves the task state while eliminating draft history that is no longer actionable.
6. **Commit checkpoints.** Have skills commit after every major step. Frequent small commits make it easy to bisect a regression introduced by an agent and revert to a known-good state without losing work.

APPENDIX B

CODEX CLI: A PRACTITIONER'S REFERENCE

Remark B.1 (Learning Objectives). After working through this appendix, the reader should be able to:

1. Install Codex CLI, authenticate it, and select the appropriate approval mode for a given task.
2. Explain the AGENTS.md file hierarchy and write project-level custom instructions for a quantitative finance codebase.
3. Describe the three kernel-level sandboxing mechanisms (Seatbelt, Landlock, seccomp) and explain when to prefer Codex CLI's OS-enforced isolation over application-layer permission controls.
4. Deploy a cloud-sandbox Codex task for processing sensitive financial documents without exposing data to the local development environment.
5. Write and install a SKILL.md file that encodes a reusable finance research workflow.
6. Compare the architectural trade-offs between Codex CLI and Claude Code and select the appropriate tool for a given research task.
7. Map at least three quantitative-finance research tasks to concrete Codex CLI workflows, including an autonomous multi-hour data pipeline.

B.1 What is Codex CLI

THE BIGGER PICTURE

On 16 April 2025, OpenAI released Codex CLI as an open-source terminal agent. The release was a deliberate competitive response: within days of Anthropic's Claude Code reaching widespread adoption, OpenAI shipped a tool with a similar surface—reads files, runs tests, edits code, commits changes—but with a fundamentally different security architecture. Where Claude Code enforces

safety at the application layer through programmable hooks and an allowlist, Codex CLI enforces safety at the operating-system kernel layer using platform sandboxing primitives. By early 2026, approximately four million developers were using Codex each week, and GPT-5.5—OpenAI's first fully agentic-first retrained base model—had become the default engine behind every Codex surface (OpenAI 2026).

B.1.1 Overview and Positioning

Codex CLI is a terminal-based agentic coding assistant that reads a repository, edits files, runs shell commands, and iterates on the results. It is the command-line surface of a broader *Codex* platform that also includes a cloud-hosted web interface, an IDE extension, a GitHub bot, and computer-use capabilities. All surfaces share the same underlying model and account context.

The most important architectural distinction from Claude Code (Appendix A) is the execution environment. Claude Code operates directly on the developer's local machine and file system; Codex CLI can optionally delegate tasks to ephemeral cloud virtual machines pre-loaded with the repository. This cloud-sandbox mode provides reproducible, isolated execution that is particularly attractive for sensitive financial data pipelines where local environment contamination or credential leakage is a concern.

B.1.2 Supported Models

As of mid-2026, Codex runs on the following model tiers:

- **GPT-5.5** — the default and recommended model; agentic-first training enables multi-hour autonomous sessions on real engineering tasks. Context window: 272K tokens (default); up to 1.05M with long-context mode enabled.
- **GPT-5.5 Pro** — higher rate limits and priority routing for teams running parallel Codex agents on large pipelines.
- **GPT-4.1** — lower cost option for high-frequency, shorter tasks such as linting, formatting, or schema validation.

Switch models with the `-model` flag or the `model` key in the configuration file. GPT-5.5's explicit agentic-first training means it is significantly better than earlier models at sustaining multi-step task state across tool calls—an important property for long financial data pipelines.

B.1.3 Key Milestones

Table B.1 summarises the major capability releases.

Table B.1. Major Codex CLI capability releases (2025–2026).

Date	Feature	Notes
April 2025	Codex CLI launch	Open-source terminal agent; three approval modes
May 2025	Cloud sandbox	Ephemeral VM execution; pre-loaded repository
September 2025	IDE extension	VS Code and JetBrains integration
October 2025	GitHub bot	`@codex` in PR comments triggers a task
December 2025	Skills support	SKILL.md spec; shared with Claude Code
February 2026	Computer use	Screen reading; desktop app interaction
April 2026	GPT-5.5 default	Agentic-first model; 1.05M max context

B.2 Installation and Setup

B.2.1 Prerequisites and Authentication

Codex CLI requires Node.js 18 or later and is installed via `npm`:

```
npm install -g @openai/codex
```

On first run, Codex opens a browser-based OAuth flow to authenticate against a ChatGPT account. For headless or CI environments, set the `OPENAI_API_KEY` environment variable:

```
export OPENAI_API_KEY="sk-..."
codex                              # start interactive session
codex "analyse the factor model in src/models/ff5.py"
```

To specify a model or run non-interactively:

```
codex --model gpt-5.5 --approval-mode full-access \
      "Refactor the CAPM implementation and add unit tests."
```

B.2.2 First-Run Configuration

On first run, Codex creates a global configuration file at `~/.codex/config.json`. The most important top-level keys are:

```
{
  "model": "gpt-5.5",
  "approvalMode": "auto",
  "projectDocMaxBytes": 32768,
  "projectDocFallbackFilenames": ["AGENTS.md", "README.md"],
  "sandbox": {
    "mode": "local",
    "networkAccess": false
  }
}
```

`projectDocMaxBytes` controls how much of the AGENTS.md file is read into context on session start (default 32 KB). For large codebases with detailed AGENTS.md conventions, increase this limit to ensure all conventions are loaded.

B.3 Core Interface and Approval Modes

B.3.1 CLI Commands and Navigation

Codex CLI's built-in slash commands are the primary navigation layer. Table B.2 lists the most commonly used commands.

Table B.2. Built-in Codex CLI slash commands.

Command	Effect
`/plan`	Produce a plan without executing any changes
`/exec`	Execute the current plan
`/review`	Review all pending changes before committing
`/permissions`	Switch approval mode interactively
`/model`	Switch the active model
`/context`	Show current context usage and limits
`/sandbox`	Toggle between local and cloud-sandbox mode
`/clear`	Start a fresh session
`/help`	List all available commands and skills

The `/plan` → `/exec` → `/review` loop is the recommended structured workflow

for any task that modifies production data or trading logic: plan in read-only mode, inspect the diff, then execute.

B.3.2 The Three Approval Modes

Approval modes govern how much Codex can do without pausing for confirmation. Unlike Claude Code's fine-grained allowlist/denylist syntax, Codex CLI offers three coarse-grained modes:

`read-only` **(Suggest mode)** Codex can browse files and propose a plan but cannot write files or run shell commands until the user approves. The safest mode for initial exploration of an unfamiliar codebase or dataset.

`auto` **(Default)** Codex can read files, edit files, and run shell commands within the working directory. It still pauses before accessing anything outside the working directory or performing network operations. Equivalent to Claude Code's Edit mode with a constrained allowlist.

`full-access` **(Danger mode)** Codex operates without confirmation across the entire machine, including network access. Use sparingly and only in sandboxed environments.

Switch modes at any time with `/permissions` or by restarting with the `-approval-mode` flag.

UNDER THE HOOD

The coarse three-mode model is a deliberate design choice. OpenAI's reasoning: most developers find fine-grained allowlists difficult to maintain correctly, so a well-enforced kernel-level boundary (see Section B.5) is a more reliable safety guarantee than an application-layer list that can be misconfigured. The trade-off is flexibility: Claude Code's hook system can express policies like "allow `git commit` but deny `git push`", whereas Codex cannot without a custom AGENTS.md guard script.

B.3.3 Context Management

GPT-5.5's default context window is 272K tokens. Long-context mode (up to 1.05M) is activated by setting `"longContext": true` in the configuration file. Key practical points:

- Use `/context` to inspect current usage before starting a long pipeline; large repositories loaded in full can exhaust the default window before the task completes.

- Unlike Claude Code's `/compact` command, Codex does not offer manual compaction. When the context limit is approached, Codex automatically summarises earlier turns.
- For pipelines that operate on many files sequentially (e.g., processing hundreds of earnings call transcripts), prefer the cloud-sandbox mode (Section B.5.2) where each task starts with a fresh context.

B.4 Configuration

B.4.1 The AGENTS.md File Hierarchy

`AGENTS.md` is Codex CLI's analogue of Claude Code's `CLAUDE.md`. It provides project-level custom instructions that Codex reads at the start of every session. The hierarchy follows the same layered pattern:

1. **Global** (`~/.codex/AGENTS.md`) — personal preferences applied to all projects.
2. **Project** (`AGENTS.md` at the repo root) — project conventions; checked into version control.
3. **Sub-directory** (any `AGENTS.md` in a nested folder) — package-level or module-level overrides.

Codex merges all three levels on startup. The `projectDocFallbackFilenames` configuration key allows additional files (e.g., `README.md`) to serve as the project instruction source when `AGENTS.md` is absent.

A representative AGENTS.md for a quantitative finance project:

```
# Project: Equity Factor Research

## Conventions
- Always use log returns: `np.log(price / price.shift(1))`.
- Date columns must be `datetime64[ns]` with timezone `UTC`.
- Factor returns are stored in `data/factors/` as Parquet files.

## Testing
- Run `pytest tests/ -x -q` after every file edit.
- Coverage must remain above 80%.

## Data Access
- Raw price data: `data/raw/prices.parquet` (read-only).
- Bloomberg data requires the `blpapi` conda environment.
```

B.4.2 Configuration Reference

Beyond the top-level keys shown in Section B.2.2, the configuration file supports:

`sandbox.mode` `"local"` (default) or `"cloud"` (ephemeral VM).

`sandbox.networkAccess` Boolean; default `false` in auto mode. Set `true` only when the task requires fetching external data.

`longContext` Boolean; enables GPT-5.5's 1.05M context window.

`historySize` Number of past sessions stored in `~/.codex/history/`; default 50.

`shell` Shell used for command execution; default `"/bin/bash"` on macOS/Linux, `"powershell"` on Windows.

B.4.3 Skills

Codex CLI implements the same open agent skills specification as Claude Code. A skill is a directory in `~/.agents/skills/` containing a required `SKILL.md` plus optional scripts and reference assets:

```
~/.agents/skills/
  fetch-factor-returns/
    SKILL.md          # name, description, invocation instructions
    scripts/
      download.py     # optional supporting script
```

Codex loads a skill when the user invokes `/skill-name` or when the task description semantically matches the skill's `description` field. Because the skill format is identical to Claude Code's, a well-written SKILL.md works in both tools without modification—a significant practical advantage for teams that use both.

B.5 Sandboxing and the Security Model

THE BIGGER PICTURE

The choice of where to enforce security boundaries has deep implications for what can go wrong when an AI agent makes a mistake or is manipulated by adversarial input. Claude Code enforces safety at the application layer: a misconfigured allowlist or a prompt-injection attack can potentially cause

the model to execute a command that the developer did not intend. Codex CLI enforces safety at the operating-system kernel layer. Even if the model requests a forbidden operation, the kernel refuses it before any harm occurs. For financial institutions with strict data governance requirements, this distinction matters.

B.5.1 Local Sandboxing

In local mode, Codex enforces sandboxing using platform-native kernel primitives:

macOS — Apple Seatbelt A mandatory access control framework built into the XNU kernel. Codex's Seatbelt profile grants read access to the working directory and system libraries, write access to the working directory only, and no outbound network access (in auto mode). The profile is applied via `sandbox-exec` and cannot be bypassed by the model or by malicious code running inside the session.

Linux — Landlock A Linux Security Module (available since kernel 5.13) that implements a stackable, unprivileged access-control mechanism (The Linux Kernel Organization 2022). Codex uses Landlock to restrict file-system access to the working directory tree.

Linux — seccomp A kernel mechanism that filters system calls. Codex's seccomp profile blocks dangerous syscalls (e.g., `execve` outside the working directory, raw socket creation) regardless of what the agent requests.

These three mechanisms work at layers below the process memory space, making them resistant to prompt injection, dependency confusion attacks, and malicious code execution via `eval`-style patterns. Table B.3 compares the two approaches.

Table B.3. Security enforcement layer: Codex CLI vs Claude Code.

Property	Codex CLI	Claude Code
Enforcement layer	OS kernel (Seatbelt/Landlock/seccomp)	Application layer
Bypass risk	Very low (kernel cannot be overridden)	Low (requires misconfigured allo
Granularity	Coarse (3 modes)	Fine (per-tool patterns)
Custom policies	Limited (AGENTS.md scripts)	Rich (allowlist + denylist syntax)
Network control	Binary (on/off per mode)	Per-command pattern matching
Platform	macOS, Linux	All platforms

B.5.2 Cloud Sandbox Execution

Setting `"sandbox.mode": "cloud"` routes each Codex task to an ephemeral virtual machine on OpenAI's infrastructure. Codex pre-loads the VM with the repository (fetched from GitHub or supplied as a tarball) and destroys it after the task completes. Benefits for financial workflows:

- **Data isolation** — sensitive datasets remain in the cloud VM and are not written to the developer's local disk.
- **Reproducibility** — each task starts from a clean environment, eliminating dependency on local Python versions, conda environments, or system libraries.
- **Parallelism** — multiple tasks can run simultaneously in separate VMs, enabling portfolio-wide analyses that would block a single local machine for hours.
- **Audit trail** — VM execution logs are retained and exportable, supporting MiFID II and similar regulatory obligations around algorithmic decision-making.

Cloud sandbox mode is activated per-session:

```
codex --sandbox cloud \
      "Run the full factor backtest suite on the 2010--2024 dataset \
       and return a JSON summary of annualised Sharpe ratios."
```

B.6 Multi-Surface Architecture

B.6.1 CLI, IDE Extension, and ChatGPT

The Codex name covers a family of surfaces that share a single model and account context:

Codex CLI The terminal agent described in this appendix; the most capable surface for autonomous, multi-step workflows.

IDE Extension Available for VS Code and JetBrains. Provides inline diffs, a plan review pane, and `@`-mention for open files, analogous to Claude Code's IDE extensions.

ChatGPT Codex The cloud-hosted version accessible through the ChatGPT interface. Delegates tasks to cloud VMs; suitable for non-technical collaborators who need to trigger data analysis tasks without a terminal.

B.6.2 GitHub Bot

Mentioning `@codex` in a GitHub pull request comment or issue triggers a Codex task that runs in a cloud sandbox against the repository at that commit. The bot can:

- write unit tests for a new function described in the PR,
- propose fixes for failing CI checks, or
- analyse whether a change to a risk model breaks any existing assumptions documented in `AGENTS.md`.

Results are posted as a follow-up PR comment with a diff and a summary.

B.6.3 Computer Use

Codex's computer-use capability (released February 2026) allows the agent to read the screen and interact with arbitrary desktop applications. For finance practitioners, this opens up workflows previously requiring manual data entry:

- extracting structured data from Bloomberg terminal screens without an API licence,
- navigating legacy proprietary trading systems to retrieve historical fills or position data, and
- filling in regulatory reporting forms that do not expose a programmatic interface.

Computer use runs in a cloud VM to prevent screen-capture of sensitive local displays.

B.7 Cost and Token Management

B.7.1 Pricing Model

Codex CLI bills at standard OpenAI API rates for the selected model. Table B.4 gives approximate consumption for representative finance research tasks using GPT-5.5.

Cloud sandbox tasks additionally incur VM compute costs billed separately by OpenAI; consult the pricing page for current rates.

B.7.2 Cost-Reduction Strategies

- Use GPT-4.1 for high-frequency, low-complexity tasks (linting, schema validation, simple reformatting); reserve GPT-5.5 for tasks requiring sustained reasoning.

Table B.4. Approximate token consumption for representative Codex CLI tasks (GPT-5.5).

Task	Input tokens	Output tokens
Explain a 500-line pricing library	8,000–15,000	1,000–3,000
Classify one earnings call transcript	10,000–20,000	1,000–2,000
Refactor a factor model module	25,000–60,000	5,000–15,000
Full backtest suite debug (multi-step)	80,000–200,000	15,000–40,000

- Enable long-context mode only when the task genuinely requires it; longer contexts incur proportionally higher input costs.
- Use cloud sandbox parallelism for bulk batch tasks (e.g., classifying 500 transcripts) rather than running them sequentially in a single long-context session.
- Store intermediate results to disk and restart sessions at natural checkpoints rather than allowing context to accumulate indefinitely.

B.8 Codex CLI in Research and Finance Workflows

B.8.1 Use Cases for Quants and Researchers

The following use cases are particularly well-suited to Codex CLI's architecture:

Large-document processing OpenAI's own finance team used Codex to review 24,771 K-1 tax forms totalling 71,637 pages, accelerating the task by two weeks relative to the prior year (OpenAI 2025). The cloud sandbox mode kept all document content within OpenAI's infrastructure and off local developer machines.

Legacy code archaeology Loading a proprietary options pricing library into a cloud sandbox and asking Codex to generate a call graph, annotate undocumented assumptions, and produce a test suite. The kernel-level sandbox prevents the legacy code from accessing network resources or writing outside the working directory during execution.

Regulatory document parsing Extracting structured data from SEC filings, ESMA reports, or FINMA circulars using the `/plan` → `/exec` structured loop, with `/review` before any database writes.

Automated backtesting Delegating overnight backtest runs to cloud sandbox VMs: each strategy variant runs in its own VM with its own fresh environment,

results are written to a shared S3 bucket, and a summary is posted to Slack on completion.

Computer-use data extraction Using the computer-use capability to extract historical volatility surface data from a Bloomberg terminal screen directly into a Parquet file, bypassing the need for a Bloomberg API licence.

B.8.2 Choosing Between Codex CLI and Claude Code

Both tools are capable general-purpose coding agents; the choice depends on the task's security and ecosystem requirements. Table B.5 summarises the key decision dimensions.

Table B.5. Decision guide: Codex CLI vs Claude Code (Appendix A).

Dimension	Codex CLI	Claude Code
Security enforcement	OS kernel (stronger guarantee)	Application layer (more flexible)
Permission granularity	3 coarse modes	Per-tool pattern matching
Cloud execution	First-class (VM sandbox)	Not built-in (CI/CD only)
Extensibility	Skills + AGENTS.md	Skills + hooks + MCP + plugins
Multi-agent	Parallel cloud VMs	Agent tool + Workflow + Agent Teams
GitHub integration	`@codex` bot	Via CI/CD non-interactive mode
Finance model access	GPT-5.5 (OpenAI ecosystem)	Claude Opus/Sonnet (Anthropic)
Context window	272K default / 1.05M max	Up to 1M (Opus 4.8)

Choose Codex CLI when:

- kernel-level sandboxing is a regulatory or security requirement,
- the task involves processing sensitive documents that must not touch a local disk,
- your team already uses OpenAI's API and adding Anthropic would create vendor complexity, or
- you need cloud-parallel execution of many independent sub-tasks.

Choose Claude Code when:

- fine-grained, per-command permission control is required (e.g., allow `git commit` but deny `git push`),
- you need MCP server integration to connect directly to Bloomberg, FactSet, or an internal data API,
- the workflow requires programmable hooks that fire on specific tool events, or

- you are building a multi-agent system with a Workflow orchestration script and Agent Teams.

B.8.3 Recommended Workflow Patterns

1. **Plan before executing.** Always start with `/plan` for any task that modifies a production dataset or trading strategy. Review the proposed diff carefully before typing `/exec`.
2. **Use cloud sandbox for sensitive data.** Configure `"sandbox.mode": "cloud"` in `~/.codex/config.json` for any project that handles NDA-covered data, client portfolios, or proprietary model weights.
3. **Encode conventions in AGENTS.md.** Every project should have a project-level `AGENTS.md` that specifies return conventions, data paths, testing requirements, and prohibited operations. Codex reads this file at session start; consistent conventions eliminate an entire class of agent errors.
4. **Separate concerns with skills.** Build a SKILL.md for each recurring workflow step—factor construction, backtest execution, report generation—and invoke them by name. Skills are portable: the same SKILL.md works in Claude Code, making it straightforward to run a workflow in either tool.
5. **Commit checkpoints frequently.** Have skills commit after each major step. Small commits let you bisect regressions introduced by the agent without losing days of pipeline work.
6. **Use `/review` before any destructive operation.** For any task that writes to a database, deletes files, or posts to an external system, issue `/review` to inspect the full diff and confirm there are no unintended side effects.

Example B.2 (End-to-end: regulatory filing extraction with Codex CLI). A compliance analyst needs to extract structured risk metrics from 200 ESMA stress-test reports (PDF) and load them into a database. Using Codex CLI in cloud sandbox mode:

```
# Configure cloud sandbox for this project (data stays off local disk)
echo '{"sandbox": {"mode": "cloud"}}' > .codex/config.json

# Step 1: Plan the extraction pipeline (read-only, no changes yet)
codex --approval-mode read-only \
  "/plan Extract risk metrics (CET1 ratio, LCR, NSFR) from PDF files in
   data/esma_reports/. Output to data/structured/esma_metrics.parquet."

# Step 2: Review the plan, then execute
```

```
codex --approval-mode auto "/exec"

# Step 3: Validate schema and commit
codex "/review"
codex "Run tests/test_esma_schema.py and confirm all 200 reports parsed."
git add data/structured/ && git commit -m "feat: ESMA stress-test metrics"
```

All PDF content is processed inside the cloud VM and never written to the analyst's local disk. The `/plan` → `/exec` → `/review` loop provides an audit trail that satisfies internal model-governance requirements.

APPENDIX C

HUGGING FACE AND LOCAL MODEL DEPLOYMENT

Remark C.1 (Learning Objectives). After working through this appendix, the reader should be able to:

1. Navigate the Hugging Face Hub to identify models and datasets suited to a given financial NLP task.
2. Load a pre-trained model with the `pipeline()` API, run inference, and interpret the output.
3. Explain the LoRA and QLoRA fine-tuning strategies, estimate the GPU memory required for a given configuration, and apply PEFT to adapt a base model to financial text.
4. Select an appropriate quantization level (GGUF Q4–Q8) for a target model size and hardware budget, and deploy the quantised model using `llama.cpp` or Ollama.
5. Deploy a production inference server with vLLM or TGI and justify the choice based on concurrency and latency requirements.
6. Articulate the data sovereignty, regulatory, and cost arguments for on-premise deployment of LLMs in regulated finance environments.
7. Design a complete local-model pipeline for a quantitative finance task—from model selection and optional fine-tuning through to a queryable REST endpoint.

C.1 The Hugging Face Ecosystem

THE BIGGER PICTURE

When Meta released LLaMA in February 2023, it uploaded the weights to the Hugging Face Hub. Within hours, the model had been downloaded tens of thousands of times. That episode crystallised the Hub's role: it had become the

de facto distribution channel for open-weight AI models, the way PyPI is for Python packages or npm for JavaScript modules. By 2026 the Hub hosted more than two million models, half a million datasets, and one million interactive demo applications. For finance practitioners, this means that virtually every open-weight language model relevant to their work—from general-purpose instruction-following models to domain-specific financial classifiers—is one `pip install` and one `from_pretrained` call away.

C.1.1 Hub: Models, Datasets, Spaces, and Endpoints

The Hugging Face Hub (`huggingface.co`) is a web platform and Python API that serves four types of artefact:

Models Pre-trained weight checkpoints stored in `safetensors` or `pytorch_model.bin` format, versioned with Git LFS. Each model card documents the training data, evaluation results, intended use, and known limitations.

Datasets Curated tabular, text, vision, and audio datasets accessible via the `datasets` library. Finance-relevant collections include earnings call transcripts, SEC filings, credit ratings histories, and financial news corpora.

Spaces Hosted Gradio or Streamlit web applications that let model authors publish interactive demos without requiring users to run code locally. Useful for evaluating a model's behaviour before committing to a download.

Inference Endpoints Managed, autoscaling API endpoints for Hub models, billed per token. They provide a production-grade alternative to self-hosting for workloads where traffic is variable or the team lacks GPU infrastructure.

Interact with the Hub programmatically via the `huggingface_hub` library:

```
from huggingface_hub import hf_hub_download, list_models

# List finance-tagged models sorted by downloads
models = list(list_models(filter="finance", sort="downloads", direction=-1))
for m in models[:5]:
    print(m.id, m.downloads)
```

C.1.2 Transformers and Datasets Libraries

The **Transformers** library (`pip install transformers`) is the primary SDK for working with Hub models. It provides:

- `AutoTokenizer`, `AutoModel`, and task-specific heads (`AutoModelForSequenceClassification`, etc.) that load any compatible model from the Hub or a local directory.
- A `pipeline()` API that wraps tokenization, inference, and post-processing into a single callable.
- `Trainer` and `SFTTrainer` for supervised fine-tuning.
- Tight integration with PEFT, bitsandbytes, and Flash Attention 2.

The **Datasets** library (`pip install datasets`) provides memory-mapped access to large datasets via Apache Arrow, enabling streaming of corpora too large to fit in RAM—an important property when fine-tuning on multi-year collections of financial filings.

```python
from datasets import load_dataset

# Stream SEC 10-K filings without downloading the full corpus
ds = load_dataset("JanosAudran/financial-reports-sec", split="train",
                  streaming=True)
for example in ds.take(3):
    print(example["text"][:200])
```

C.2 Model Discovery and the Pipeline API

C.2.1 Browsing the Hub for Finance Tasks

The Hub's filter system lets practitioners narrow to models by task, language, and licence. For finance NLP the most relevant task tags are:

`text-classification` Sentiment analysis, intent classification, regulatory document categorisation.

`token-classification` Named entity recognition for financial entities (companies, currencies, financial instruments).

`question-answering` Extractive QA over earnings reports and analyst transcripts.

`text-generation` Instruction-following models for report drafting, data extraction, and chain-of-thought reasoning over financial data.

`feature-extraction` Embedding models for semantic search over document corpora and RAG pipelines.

C.2.2 The Pipeline API

The `pipeline()` function is the quickest path from Hub model to inference result:

```
from transformers import pipeline

# Zero-shot sentiment on an earnings call excerpt
clf = pipeline("zero-shot-classification",
               model="facebook/bart-large-mnli",
               device=0)          # device=0 → GPU; -1 → CPU

labels = ["positive", "negative", "neutral"]
result = clf(
    "Revenue grew 18% year-over-year, driven by strong mortgage originations.",
    candidate_labels=labels,
)
# {'labels': ['positive', 'neutral', 'negative'], 'scores': [0.89, 0.08, 0.03]}
print(result)
```

For generation tasks, set `max_new_tokens` and `do_sample=False` for deterministic output—important in research contexts where reproducibility matters:

```
gen = pipeline("text-generation", model="mistralai/Mistral-7B-Instruct-v0.3",
               device_map="auto", torch_dtype="auto")

output = gen(
    "<s>[INST] Summarise the key risk factors from: {text} [/INST]",
    max_new_tokens=256, do_sample=False,
)
print(output[0]["generated_text"])
```

C.2.3 Finance-Specific Model Families

Several model families are specifically relevant to quantitative finance:

FinBERT (Araci 2019) A BERT model pre-trained on financial communication and fine-tuned for financial sentiment classification. Available at `ProsusAI/finbert` on the Hub. The standard baseline for earnings call and news sentiment tasks.

BloombergGPT (Wu et al. 2023) A 50-billion-parameter model trained on Bloomberg's proprietary financial corpus. Not publicly available on the Hub but provides the benchmark against which open-weight finance models are evaluated.

FinLLMs A family of open-weight models fine-tuned on financial instruction datasets; evaluated on FinQA, FPB (Financial Phrase Bank), and Headline classification benchmarks. Multiple variants at different parameter scales are available on the Hub.

Llama 3 / Mistral 7B instruction variants General-purpose instruction- following models that, with appropriate prompting or light fine-tuning, perform competitively on financial NLP tasks while remaining locally deployable on commodity GPU hardware.

Table C.1 gives a concise evaluation snapshot across key finance NLP benchmarks.

Table C.1. Selected model performance on finance NLP benchmarks (higher is better).

Model	Size	FPB F1	FinQA	Headline
FinBERT	110M	0.88	—	0.97
Llama 3.1 8B (inst.)	8B	0.82	0.61	0.93
Mistral 7B (inst.)	7B	0.80	0.58	0.91
BloombergGPT	50B	0.87	0.68	0.97

FPB: Financial Phrase Bank. FinQA: numerical reasoning over financial tables. Headline: financial news headline classification. Scores are approximate.

C.3 Parameter-Efficient Fine-Tuning

THE BIGGER PICTURE

Full fine-tuning a 7-billion-parameter model requires updating all seven billion weights simultaneously, which at 16-bit precision demands roughly 14 GB of GPU memory for the weights alone. Optimizer states add substantially more: AdamW stores a float32 master copy of the weights plus first and second moment tensors, totalling at least 112 GB for a 7B model in mixed-precision training. For most practitioners, this is infeasible. Parameter-efficient fine-tuning (PEFT) methods solve this by freezing the pre-trained weights and learning a small number of additional parameters that steer the model's behaviour. LoRA reduces the memory requirement by a factor of 10–20 while retaining 90–95% of the quality of full fine-tuning, enabling adaptation on a single consumer GPU.

C.3.1 LoRA: Low-Rank Adaptation

LoRA (Hu et al. 2022) decomposes each weight update $\Delta W \in \mathbb{R}^{d \times k}$ as the product of two low-rank matrices:

$$\Delta W = BA, \quad B \in \mathbb{R}^{d \times r},\ A \in \mathbb{R}^{r \times k},\ r \ll \min(d, k). \tag{C.1}$$

During training only A and B are updated; the pre-trained weight W_0 is frozen. At inference, the adapter can be merged: $W = W_0 + \Delta W$, adding zero latency overhead. The rank r is a hyperparameter: values of 4–16 are typical for fine-tuning general instruction-following models; higher ranks (32–64) are used when the target domain is very different from the pre-training distribution.

```python
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import get_peft_model, LoraConfig, TaskType

base = AutoModelForCausalLM.from_pretrained("mistralai/Mistral-7B-v0.3",
                                            torch_dtype="auto",
                                            device_map="auto")

config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=16,                          # rank
    lora_alpha=32,                 # scaling factor
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
)

model = get_peft_model(base, config)
model.print_trainable_parameters()
# trainable params: 6,815,744 || all params: 7,248,547,840 || trainable%: 0.09
```

C.3.2 QLoRA: Quantised Low-Rank Adaptation

QLoRA (Dettmers et al. 2023) combines LoRA with 4-bit NormalFloat (NF4) quantisation of the frozen base weights. This reduces the memory footprint of a 7B model from approximately 14 GB (bfloat16) to approximately 4–5 GB, making fine-tuning feasible on a single consumer GPU with 6–8 GB of VRAM.

```python
from transformers import BitsAndBytesConfig
import torch

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
```

```
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True,    # nested quantization: saves ~0.4 GB
)

base = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-v0.3",
    quantization_config=bnb_config,
    device_map="auto",
)
```

C.3.3 Fine-Tuning on Financial Text

A practical fine-tuning pipeline for a financial instruction dataset using the `SFTTrainer` from the TRL library:

```
from trl import SFTTrainer
from transformers import TrainingArguments
from datasets import load_dataset

dataset = load_dataset("json",
                       data_files="data/finance_instruct.jsonl",
                       split="train")

args = TrainingArguments(
    output_dir="checkpoints/mistral-finance",
    num_train_epochs=3,
    per_device_train_batch_size=2,
    gradient_accumulation_steps=8,
    learning_rate=2e-4,
    fp16=True,
    logging_steps=10,
    save_strategy="epoch",
)

trainer = SFTTrainer(
    model=model,          # QLoRA model from previous step
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=2048,
    args=args,
)
trainer.train()
```

Merging the adapter back into the base weights for deployment:

```
from peft import PeftModel

merged = PeftModel.from_pretrained(base, "checkpoints/mistral-finance/final")
merged = merged.merge_and_unload()
merged.save_pretrained("models/mistral-finance-merged")
```

C.4 Quantization and the GGUF Format

C.4.1 Quantization Levels and Quality Trade-offs

Post-training quantization compresses model weights from 16-bit floats to lower-bit integer representations, reducing both memory footprint and inference latency. Table C.2 summarises the commonly used levels in the GGUF ecosystem.

Table C.2. GGUF quantization levels: memory and quality trade-offs for a 7B model.

Quantization	VRAM (7B)	Quality loss	Best use
Q2_K	≈2.8 GB	High (>5%)	Ultra-constrained hardware only
Q4_K_M	≈4.1 GB	Low (<1%)	Standard local deployment
Q5_K_M	≈4.8 GB	Very low	Accuracy-sensitive tasks
Q6_K	≈5.5 GB	Negligible	Near-lossless on 6 GB VRAM
Q8_0	≈7.2 GB	Negligible	Research: maximal quality
F16	≈14 GB	None (baseline)	Serving; requires server GPU

For most finance NLP tasks, Q4_K_M delivers the best balance: approximately 75% memory reduction relative to full precision with less than 1% quality degradation on standard benchmarks (Gerganov et al. 2023).

C.4.2 GGUF and llama.cpp

GGUF (GPT-Generated Unified Format) is the standard single-file format for quantized model checkpoints. A GGUF file bundles the quantized weights, the tokenizer vocabulary, positional encoding parameters, and all metadata needed to run inference—enabling distribution as a single artefact with no separate configuration files.

`llama.cpp` is an open-source C/C++ inference engine that reads GGUF files and runs on CPUs, Apple Silicon (via Metal), and NVIDIA GPUs (via CUDA). It is the reference implementation for local LLM inference and underpins most higher-level tools, including Ollama.

```
# Download a GGUF model from the Hub (bartowski provides many quantizations)
huggingface-cli download bartowski/Mistral-7B-Instruct-v0.3-GGUF \
    --include "Mistral-7B-Instruct-v0.3-Q4_K_M.gguf" \
    --local-dir models/

# Run interactive inference
./llama-cli -m models/Mistral-7B-Instruct-v0.3-Q4_K_M.gguf \
            -n 256 -p "Summarise the credit risk in: {text}"
```

`llama.cpp` also exposes an OpenAI-compatible HTTP server:

```
./llama-server -m models/Mistral-7B-Instruct-v0.3-Q4_K_M.gguf \
               --port 8080 --n-gpu-layers 35
```

UNDER THE HOOD

The `-n-gpu-layers` flag controls how many transformer layers are offloaded to the GPU; the remainder run on CPU. For a 7B model with 32 layers on a machine with 4 GB of GPU VRAM, offloading 20–24 layers typically maximises throughput while keeping the remainder in system RAM. The optimal split is model- and hardware-specific; benchmark with `llama-bench` before committing to a production configuration.

C.4.3 Hardware Requirements by Model Size

Table C.3 gives the minimum GPU VRAM needed to run inference at Q4_K_M quantization for the most commonly deployed open-weight model sizes.

Table C.3. Minimum GPU VRAM for inference at Q4_K_M quantization.

Model size	VRAM (Q4_K_M)	Example models	Consumer GPU
3–4B	≈2.5 GB	Phi-3 Mini, Llama 3.2 3B	RTX 3060 (8 GB)
7–8B	≈4.5 GB	Mistral 7B, Llama 3.1 8B	RTX 3060 (8 GB)
13B	≈8 GB	Llama 2 13B, CodeLlama 13B	RTX 3080/4070
32–34B	≈20 GB	Llama 3.3 32B, CodeLlama 34B	RTX 4090 (24 GB)
70B	≈42 GB	Llama 3.1 70B	2× RTX 4090 or A100

C.5 Local Deployment with Ollama

Ollama provides a user-friendly wrapper around llama.cpp that manages model downloads, versioning, and a persistent OpenAI-compatible REST server. By early 2026 it had accumulated over 112 million pulls for Llama 3.1 alone, making it the most widely used local LLM runtime in the developer community.

C.5.1 Installation and Model Management

```
# Install (macOS/Linux)
curl -fsSL https://ollama.com/install.sh | sh

# Pull and run a model (starts the server automatically)
ollama pull mistral                # downloads Mistral 7B Q4_K_M by default
ollama run mistral "Explain duration risk in plain language."

# List locally available models
ollama list
```

Ollama stores models in `~/.ollama/models/` and manages multiple versions side by side. Custom GGUF files can be registered with a `Modelfile`:

```
# Modelfile for a finance-tuned variant
FROM ./models/mistral-finance-merged.gguf
SYSTEM "You are a quantitative finance assistant. Always use log returns and
        cite data sources."
PARAMETER temperature 0.1
PARAMETER num_ctx 8192
```

```
ollama create mistral-finance -f Modelfile
ollama run mistral-finance "What is the Sharpe ratio of this return series?"
```

C.5.2 REST API and Python Integration

Ollama exposes an OpenAI-compatible API at `http://localhost:11434`, enabling drop-in replacement for OpenAI API calls:

```
from openai import OpenAI          # standard openai client

```

```python
client = OpenAI(
    base_url="http://localhost:11434/v1",
    api_key="ollama",              # required but ignored by Ollama
)

response = client.chat.completions.create(
    model="mistral-finance",
    messages=[
        {"role": "system", "content": "You are a credit risk analyst."},
        {"role": "user",   "content": "Summarise the key covenants in: ..."},
    ],
    temperature=0.0,
)
print(response.choices[0].message.content)
```

Existing Python code that calls the OpenAI API can be redirected to a local Ollama instance by changing two lines—`base_url` and `api_key`—with no other modifications.

C.6 Production Inference Servers

THE BIGGER PICTURE

Single-user local inference with Ollama or llama.cpp is sufficient for individual research tasks. When a model must serve multiple analysts simultaneously—routing queries from a Slack bot, a web application, or an automated pipeline—a production inference server is required. The two dominant open-source options are Hugging Face's Text Generation Inference (TGI) and the community-developed vLLM. They are architecturally distinct and suit different workload profiles.

C.6.1 Text Generation Inference (TGI)

TGI is Hugging Face's production-grade inference server and the backend for Hugging Face Inference Endpoints. Its design prioritises low tail latency in interactive, single-user scenarios. Key features:

- Flash Attention 2 and Paged Attention for memory-efficient attention.
- Continuous batching, enabling multiple concurrent requests to share a single forward pass.
- Native support for AWQ, GPTQ, and bitsandbytes quantization.

- OpenAI-compatible API with streaming support.

```
docker run --gpus all \
  -p 8080:80 \
  -v $HOME/models:/data \
  ghcr.io/huggingface/text-generation-inference:latest \
  --model-id mistralai/Mistral-7B-Instruct-v0.3 \
  --quantize bitsandbytes-nf4 \
  --max-total-tokens 4096
```

C.6.2 vLLM and PagedAttention

vLLM (Kwon et al. 2023) introduces *PagedAttention*, an attention mechanism inspired by virtual memory paging. Rather than allocating a contiguous block of GPU memory for each request's KV cache, PagedAttention stores KV cache pages in non-contiguous blocks and maps them on demand. This eliminates memory fragmentation, enables much larger effective batch sizes, and delivers up to 24× higher throughput than TGI under high-concurrency workloads.

```
pip install vllm

python -m vllm.entrypoints.openai.api_server \
    --model mistralai/Mistral-7B-Instruct-v0.3 \
    --quantization awq \
    --max-model-len 8192 \
    --port 8000
```

In production, Stripe achieved a 73% inference cost reduction by migrating to vLLM, processing 50 million daily API calls on one-third of the original GPU fleet (Kwon et al. 2023).

C.6.3 Choosing an Inference Server

For quantitative finance workflows, vLLM is typically the better choice when running batch sentiment analysis, document extraction pipelines, or overnight model monitoring jobs. TGI is preferable for interactive analyst tools where individual query latency matters more than aggregate throughput.

Table C.4. Inference server selection guide: TGI vs vLLM.

Criterion	TGI (Hugging Face)	vLLM
Throughput (high concurrency)	Good	Excellent (up to 24× TGI)
Tail latency (single user)	Excellent	Good
Quantization support	AWQ, GPTQ, BnB	AWQ, GPTQ, FP8
Kubernetes / Helm charts	Official support	Community Helm chart
HF Hub model compatibility	Native	Via `-model` flag
Best fit	Interactive apps	Batch / high-throughput API

C.7 Privacy, Compliance, and On-Premise Deployment

C.7.1 Data Sovereignty and Regulatory Requirements

Cloud-hosted LLM APIs transmit prompt text to a third-party server. For financial institutions, this raises several regulatory concerns:

- **Data residency** — MiFID II, DORA, and national data protection laws may prohibit transferring client or trading data outside specified jurisdictions.
- **Model risk governance** — SR 11-7 and equivalent frameworks require auditability of every model that influences a financial decision; a third-party API's internals are opaque.
- **Confidential information** — Analyst reports, pending M&A data, and portfolio holdings are material non-public information; submitting them to an external API may violate insider-trading regulations.

Local and on-premise deployment eliminates the data-transmission risk entirely: all prompts and completions remain within the institution's controlled environment.

C.7.2 On-Premise Cost Analysis

On-premise GPU servers have high upfront capital costs but dramatically lower per-token compute costs for sustained workloads. A server equipped with 8× NVIDIA H100 GPUs costs approximately $0.87 per hour in electricity and cooling; the equivalent cloud configuration costs approximately $98.32 per hour, a 113× difference (OpenAI 2025). For a team running continuous inference workloads, the breakeven point against cloud API pricing is typically reached within 6–18 months of acquisition.

Table C.5. On-premise vs cloud cost comparison for sustained LLM inference.

Cost component	On-premise (8× H100)	Cloud (8× H100)
Compute (per hour)	$0.87	$98.32
Capital amortisation	$3.50	—
Effective total/hour	$4.37	$98.32
Annual cost (24/7)	$38,281	$861,283

C.7.3 Architecture Patterns for Finance

Three deployment patterns are common in regulated finance environments:

Air-gapped on-premise All model weights, serving infrastructure, and data reside within the institution's data centre. Maximum security; highest operational burden. Required for most tier-1 trading system use cases.

Private cloud (VPC) Model weights are loaded into a dedicated tenant environment on a major cloud provider's infrastructure. Data remains within the institution's VPC; regulatory requirements for data residency are typically satisfied.

Hybrid Public API for non-sensitive tasks (e.g., public document summarisation, general code assistance); on-premise or VPC deployment for tasks involving client or trading data. Most cost-effective for institutions with a mix of sensitivity levels across workflows.

C.8 Finance Research Workflows

C.8.1 Use Cases for Quants and Researchers

Earnings call sentiment Run FinBERT or a fine-tuned Mistral variant over quarterly earnings transcripts to construct a sentiment factor. A Q4_K_M quantised model running on a workstation GPU can process 500 transcripts per hour without any API cost.

Financial document extraction Use a structured-output pipeline to extract numerical tables, covenant definitions, or risk factor summaries from PDF filings. Local deployment eliminates concerns about disclosing confidential deal terms to a cloud provider.

RAG over internal research Build a retrieval-augmented generation system over internal research notes using a local embedding model (e.g., `BAAI/bge-m3`) and

a local generation model. Queries and responses never leave the institution's network.

Domain-adapted fine-tuning Use QLoRA to adapt a 7B general model to the vocabulary and reasoning style of regulatory filings, credit agreements, or derivatives term sheets. A 3-epoch fine-tune on 10,000 examples completes in 4–6 hours on a single RTX 4090.

Stress testing and scenario generation Prompt a local instruct model to generate hypothetical market narratives or macroeconomic scenarios for stress-testing model assumptions.

Example C.2 (End-to-end: local earnings-call sentiment factor). A researcher builds a monthly sentiment factor from quarterly earnings calls using a fully local pipeline, with no data leaving the workstation.

```python
from transformers import pipeline
from datasets import Dataset
import pandas as pd

# 1. Load FinBERT once; all inference is local
clf = pipeline("text-classification",
               model="ProsusAI/finbert",
               device=0,           # GPU
               truncation=True, max_length=512)

# 2. Load earnings call transcripts (local CSV)
df = pd.read_csv("data/transcripts_2010_2024.csv")  # ticker, date, text

# 3. Batch inference (HF datasets handles batching automatically)
hf_ds = Dataset.from_pandas(df[["text"]])
results = hf_ds.map(
    lambda batch: {"label": [r["label"] for r in clf(batch["text"])],
                   "score": [r["score"]  for r in clf(batch["text"])]},
    batched=True, batch_size=32,
)

df["sentiment"] = results["label"]
df["confidence"] = results["score"]

# 4. Map label to numeric score, compute monthly averages
score_map = {"positive": 1, "neutral": 0, "negative": -1}
df["score_num"] = df["sentiment"].map(score_map)
factor = df.groupby(["ticker", pd.Grouper(key="date",
    freq="ME")])["score_num"].mean()

factor.to_csv("data/factors/finbert_sentiment.csv")
```

```
print(factor.describe())
```

The entire pipeline runs on-premise, processes 500 transcripts per hour on a single RTX 3080, and produces a factor ready for integration into a Fama–French regression.

C.8.2 Recommended Workflow Patterns

1. **Start with the Hub, then localise.** Prototype with the `pipeline()` API against the cloud Hub to identify the right model and prompt format. Once the approach is validated, download the GGUF or safetensors weights and switch to local inference.
2. **Match quantization to task sensitivity.** Use Q4_K_M for classification and extraction tasks where 1% quality loss is acceptable. Use Q6_K or F16 for tasks requiring precise numerical reasoning (e.g., extracting financial ratios from tables).
3. **Fine-tune on domain data before deploying at scale.** A QLoRA adapter trained on 5,000–10,000 finance-domain examples typically outperforms a much larger general model on domain-specific tasks at a fraction of the inference cost.
4. **Use vLLM for batch, Ollama for interactive.** Route overnight bulk pipelines through vLLM's OpenAI-compatible API; route analyst interactive queries through Ollama. Both expose the same API schema, so switching requires only a `base_url` change.
5. **Enforce data-handling rules via the inference server.** Set hard limits on `max_tokens` and log all requests server-side. Never pass raw client data through a model server that is not subject to your institution's data-retention and access-control policies.
6. **Version your models like code.** Store model checkpoints and GGUF files in a content-addressed registry (e.g., DVC or MLflow artifacts). Tag each deployment with the model hash, quantization level, and fine-tuning dataset version so that results are reproducible months later.

APPENDIX D

BUILDING COMMERCIAL APPLICATIONS WITH THE ANTHROPIC SDK AND CLAUDE CODE SDK

Remark D.1 (Learning Objectives). After working through this appendix, the reader should be able to:

1. Explain when the interactive Claude Code CLI is insufficient and a programmatic SDK approach is required for commercial deployments.

2. Install the Anthropic Python or TypeScript SDK, authenticate with an API key, and make a basic completion call with the correct parameters.

3. Define tools as JSON schemas, handle tool-call responses in a loop, and extract structured outputs using the SDK.

4. Implement a self-contained agentic loop that manages conversation state, calls tools, and handles errors and retries without human input.

5. Spawn and manage Claude Code processes headlessly via the Claude Code SDK, capturing structured output for downstream pipeline steps.

6. Design a multi-agent workflow that fans work out in parallel, aggregates results, and passes state between orchestrator and subagents.

7. Integrate token streaming into a web-facing API endpoint and handle partial response buffering correctly.

8. Apply production-engineering practices—rate-limit handling, exponential backoff, cost budgeting, and structured logging—to an SDK-based service.

9. Identify the security and compliance requirements specific to financial applications (API key hygiene, PII handling, audit trails, HITL controls).

10. Trace through a complete automated financial report generator built on the Anthropic SDK, identifying where each design principle is applied.

D.1 When to Go Beyond the Interactive CLI

The interactive Claude Code CLI, described in Appendix A, occupies a well-defined niche: it is a force-multiplier for the individual developer who needs to explore a codebase, draft a chapter, or run a one-off data analysis without leaving the terminal. Its conversational interface, built-in tool integrations, and real-time display of reasoning make it an exceptionally productive environment for tasks that benefit from human steering. For that use case, it is close to optimal.

Commercial software products operate under fundamentally different constraints. A fintech SaaS platform must serve thousands of concurrent users from a single code-base. A research data pipeline must run nightly without human supervision, producing reports that feed downstream risk systems. A real-time trading desk tool must respond in under a second to incoming market data. None of these scenarios can be built on top of an interactive REPL. They require programmatic, headless, scalable access to Claude's capabilities—access provided by the Anthropic SDK and, for tasks that demand Claude Code's full agentic toolkit, the Claude Code SDK (Anthropic 2025a; Anthropic 2025b).

This appendix introduces both interfaces. It begins with the limitations of the interactive model for production systems, moves through the Anthropic SDK's core primitives (messages, tool use, streaming), builds up to multi-agent orchestration patterns, and concludes with the production-engineering and compliance concerns that any commercial deployment in the financial industry must address. Throughout, code examples are grounded in realistic financial use cases—earnings analysis, equity research automation, and report generation—so that the architectural patterns are immediately applicable.

D.1.1 Limitations of the Interactive Model for Production Systems

The interactive CLI is, by design, a human-in-the-loop tool. Every invocation opens a terminal session, renders output to a REPL, and expects a human to read and respond. This design is a virtue for exploratory work and a fundamental constraint for automation. Several concrete limitations emerge when one attempts to stretch the CLI into a production role.

First, there is no headless operation mode. The CLI requires an attached terminal and, implicitly, a human operator who can interpret its output and provide follow-up instructions. Batch jobs, web servers, and microservices run without any attached terminal; they must be invoked programmatically and must return results in a machine-readable format. The CLI's primary output channel is a rich, human-readable display that is not designed for downstream parsing.

Second, the CLI provides no mechanism for programmatic control over conversa-

tion state or tool execution. A production system must be able to inject context into a prompt dynamically (the current user's identity, account permissions, real-time market data), route tool calls to specific implementations, intercept errors, and retry failed operations—all without human input. The CLI abstracts these concerns away from the user, but that abstraction becomes an obstacle when the system itself needs to be the user.

Third, the interactive model cannot be embedded in multi-tenant web applications, API servers, or CI/CD pipelines without resorting to fragile subprocess hacks that bypass the CLI's output parsing, capture its stdout, and inject prompts through its stdin. Such wrappers are brittle, hard to test, and carry no guarantees about compatibility across CLI versions. The SDK, by contrast, is versioned, typed, and designed specifically for programmatic use.

THE BIGGER PICTURE

A solo researcher running `/analyse earnings.pdf` to extract key metrics opens a terminal, types a prompt, reads the response, and decides what to do next. The entire workflow takes thirty seconds and requires one human. A fintech SaaS platform serving ten thousand analysts cannot work this way. Each of those analysts submits requests through a browser; the backend must call Claude, stream the response to the client, log the exchange for compliance, and attribute the cost to the correct billing account, all within a single HTTP request lifecycle. The interactive model is not the wrong tool because it is bad; it is the wrong tool because it was designed for a different job. The SDK exists precisely to fill this gap.

Fourth, the CLI offers no per-user isolation, cost attribution, or multi-tenancy. In a commercial deployment, every API call must be associated with a user identity and an organisational billing account. Rate limits must be enforced at the application layer. Token spend must be tracked, capped, and reported. None of these concerns can be addressed by the CLI without building the very abstraction layer that the SDK already provides.

Finally, integration with CI/CD systems—automated test pipelines, pull request review bots, documentation generators—requires that Claude be callable as a library function returning a Python object or a TypeScript promise, not as a process whose output is printed to a terminal. The SDK enables this integration pattern natively, while the CLI requires subprocess plumbing that is fragile by comparison.

D.1.2 Taxonomy of Programmatic Use Cases

Definition D.2 (Programmatic LLM Integration). A *programmatic LLM integration* is any system in which calls to a large language model are initiated by application code—not by a human interacting with a conversational interface—and in which the model's responses are consumed by subsequent code rather than being presented directly to a human for interpretation.

Programmatic integrations span a wide range of architectural patterns. The following taxonomy, while not exhaustive, covers the principal use cases encountered in financial applications.

1. **SaaS applications with embedded AI.** A web or mobile application integrates Claude to power features such as document summarisation, Q&A over financial filings, or AI-assisted portfolio commentary. The application backend calls the SDK, streams results to the frontend, and bills the cost per user session.
2. **Batch document processing pipelines.** A nightly job ingests hundreds of earnings transcripts, analyst reports, or regulatory filings, extracts structured data from each (revenue figures, forward guidance, risk factors), and loads the results into a data warehouse. Throughput and cost efficiency matter more than latency; the Haiku model tier is often appropriate here.
3. **Automated research agents.** An agent loop retrieves financial data through tool calls, reasons over it across multiple turns, and produces a formatted research note. Unlike a batch pipeline, the agent's path through the task is not predetermined: it adapts based on what data it finds.
4. **Real-time financial data analysis.** A market data feed triggers a Claude call whenever a watchlist stock moves more than two standard deviations in five minutes. The model's output—a one-paragraph situational summary—is pushed to a trader's dashboard within seconds. Streaming is essential here.
5. **Internal tooling and APIs.** An internal REST API wraps Claude and exposes it to other teams as a service, with access controls, rate limits, and audit logging managed centrally. Individual teams consume the service without managing their own API keys or model parameters.
6. **CI/CD code generation and review hooks.** A pre-merge hook calls Claude to review a pull request for compliance with internal coding standards, or to generate docstrings for new functions. The hook runs headlessly in a container, posts comments to the version control system, and fails the pipeline if critical issues are found.

Each of these patterns places different demands on the SDK. Batch pipelines prioritise throughput and cost; real-time dashboards prioritise latency and stream-

ing; SaaS applications require multi-tenancy and per-user cost attribution; CI/CD hooks require determinism and structured output. The remainder of this appendix develops the primitives and patterns that serve all of these needs.

D.2 The Anthropic SDK

The Anthropic SDK is the primary programmatic interface to all Claude models. It is available as a Python package (`anthropic`) and a TypeScript/JavaScript package (`@anthropic-ai/sdk`). Both packages wrap the same underlying HTTP API (Anthropic 2025a), providing authentication, request serialisation, response parsing, streaming support, automatic retries with exponential back-off, and typed response objects that make it impossible to misread a field name at runtime.

The Python SDK is the natural choice for financial data science and research applications, where the surrounding ecosystem—NumPy, pandas, scikit-learn, and similar libraries—is predominantly Python. The TypeScript SDK is the right choice for Node.js web servers, serverless functions, and any frontend-adjacent code that runs in a JavaScript runtime. Throughout this appendix, primary examples are given in Python with TypeScript equivalents noted where the idioms differ meaningfully.

The SDK abstracts away several concerns that would otherwise require substantial boilerplate. Retry logic with jitter, connection pooling, chunked response parsing for streaming events, and the translation of HTTP error codes into typed Python exceptions are all handled automatically. A developer using the SDK can focus entirely on the logic of their application rather than on the mechanics of the transport layer.

D.2.1 Installation and Authentication

Installation follows the standard package manager conventions for each ecosystem. In Python, the SDK is distributed on PyPI:

```
pip install anthropic
```

In Node.js and TypeScript environments, it is distributed on npm:

```
npm install @anthropic-ai/sdk
```

Both packages require no native extensions; they are pure Python and pure JavaScript respectively, which means they install without compilation on any plat-

form and work inside virtual environments, Docker containers, and serverless runtimes without modification.

Authentication is managed through an API key issued from the Anthropic console. The canonical pattern is to supply the key through the `ANTHROPIC_API_KEY` environment variable, which both SDKs read automatically when no explicit key is provided:

```
import anthropic

# The client reads ANTHROPIC_API_KEY from the environment automatically.
# Do not pass the key as a literal string in source code.
client = anthropic.Anthropic()
```

The injunction against hardcoding keys in source code is not merely a best practice; it is a security requirement. API keys embedded in source files are trivially exposed through version control history, container image layers, and log output. For local development, store the key in a `.env` file that is listed in `.gitignore` and load it with a library such as `python-dotenv`. In staging and production environments, retrieve the key from a secrets manager—AWS Secrets Manager, HashiCorp Vault, or GCP Secret Manager—at application startup and inject it into the process environment. Never log the key, never pass it as a command-line argument, and never include it in error messages.

```
# Local development pattern using python-dotenv
from dotenv import load_dotenv
import os, anthropic

load_dotenv()  # reads .env into os.environ
client = anthropic.Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"])
```

The `Anthropic` constructor also accepts explicit `base_url`, `max_retries`, and `timeout` parameters, which are covered in Section D.8.1.

D.2.2 Making Your First API Call

The core abstraction in the Anthropic API is the *Messages* endpoint. A call to `client.messages.create` sends a list of conversational turns to the model and returns a `Message` response object. The minimal parameters are `model`, `max_tokens`, and `messages`; the optional `system` parameter provides a persistent instruction that frames the model's behaviour for the entire conversation.

The following example asks Claude to distil the investment thesis for a hypothetical company into a single sentence, a common operation in earnings analysis pipelines:

```python
import anthropic

client = anthropic.Anthropic()

message = client.messages.create(
    model="claude-opus-4-8",
    max_tokens=256,
    system="You are a financial analyst specialising in equity research.",
    messages=[
        {
            "role": "user",
            "content": (
                "In one sentence, summarise the investment thesis for a company "
                "reporting 20% revenue growth, expanding margins, and raising
                 ↪ guidance."
            ),
        }
    ],
)

# The response text lives inside the first content block.
print(message.content[0].text)
print(
    f"Tokens used: {message.usage.input_tokens} in "
    f"/ {message.usage.output_tokens} out"
)
```

The `Message` object returned by `create` has several fields that production code regularly inspects. `message.content` is a list of content blocks; for a simple text response there is exactly one block, an instance of `TextBlock`, whose `.text` attribute holds the generated string. `message.stop_reason` reports why generation halted: `"end_turn"` means the model chose to stop; `"max_tokens"` means the `max_tokens` limit was reached (a signal to increase the limit or split the request); `"tool_use"` means the model emitted a tool call requiring application-side execution. `message.usage` is a `Usage` object with `input_tokens` and `output_tokens` fields, both of which feed directly into cost accounting.

Example D.3 (Typical token expenditure for financial summaries). A system prompt of 30 tokens plus a user message of 50 tokens totals 80 input tokens. At the `claude-opus-4-8` rate of $15 per million input tokens, this costs $0.0000012 per call. One million such calls—a plausible daily volume for a high-traffic fintech

SaaS—cost $1.20 in input tokens plus output token costs depending on response length. At 50 output tokens per call and $75 per million output tokens, the output cost is $3.75 per million calls. Total cost per million calls is approximately $4.95, illustrating why model selection matters at scale.

D.2.3 Model Selection and Parameters

The Anthropic model family is organised into three capability tiers, each optimised for a different point on the speed–cost–quality trade-off curve. Table D.1 summarises the key characteristics.

Table D.1. Anthropic model tiers and indicative pricing (as of 2026).

Model ID	Tier	Input ($/M tok)	Output ($/M tok)	Best suited for
`claude-haiku-4-5-20251001`	Haiku	0.25	1.25	Classification, ta entity extraction, volume batch jobs
`claude-sonnet-4-6`	Sonnet	3.00	15.00	Analysis, summ tion, code gene streaming chat
`claude-opus-4-8`	Opus	15.00	75.00	Complex multi-st soning, long-conte thesis, research ag

Beyond `model` and `max_tokens`, the most consequential parameters are `system`, `temperature`, and `stop_sequences`. The system prompt establishes the model's role, constraints, output format, and any domain knowledge that should be assumed throughout the conversation; it is the principal lever for shaping response behaviour and should be crafted with the same care as a software interface contract. The `temperature` parameter, which ranges from zero to one with a default of one, controls the stochasticity of token sampling: higher values produce more varied and creative outputs, while lower values concentrate probability mass on the most likely tokens. In financial applications, determinism is often more valuable than creativity; setting `temperature=0` in production pipelines produces consistent outputs that are easier to test and audit.

Remark D.4. Setting `temperature=0` does not guarantee bit-for-bit identical outputs across all runs, because the model may still encounter rounding differences at the hardware level across different inference hosts. For tasks that require absolute reproducibility (regression testing, audit trails), record the full response including the `model` field and timestamp alongside the output. Do not rely on identical token

sequences; rely instead on semantic equivalence validated by a downstream parser or schema check.

The `stop_sequences` parameter accepts a list of strings; generation halts as soon as any of them appears in the output. This is useful for extracting the first sentence of a response or for terminating generation once a delimiter such as `"--"` is emitted. In multi-turn agent loops, stop sequences can signal the model to pause and wait for tool results rather than continuing to hallucinate downstream steps.

For tasks that are clearly within the Haiku tier's capabilities—single-label classification, named-entity tagging, boolean yes/no judgements—the cost savings relative to Opus are dramatic: at equal token counts, Haiku input tokens cost sixty times less than Opus input tokens. A batch job that processes ten million earnings transcript sentences costs $2.50 in Haiku input tokens versus $150 in Opus input tokens. The discipline of matching model tier to task complexity is one of the highest- leverage cost-reduction levers available to a production engineering team.

D.3 Tool Use and Structured Outputs

The Messages API supports a mechanism by which the model can emit structured calls to user-defined functions rather than generating free-form text. This capability, sometimes called function calling or tool use, was studied empirically in the context of general-purpose language models by Schick et al. (2023), who showed that models can learn to insert API calls inline with generated text, and later extended to the domain of API-calling by Patil et al. (2023), who demonstrated that models can select correctly among thousands of real-world API signatures given natural language descriptions. The Anthropic implementation follows the same conceptual pattern: the developer declares a set of tools as JSON schemas, the model decides when to invoke them based on the user's request, and the application executes the tool and returns the result for the model to incorporate into its next response.

This mechanism is the foundation for building agents that interact with external systems. A financial application might give Claude access to a data warehouse query function, a market data feed, a portfolio management API, and a document retrieval system. The model then acts as an intelligent router, selecting the right tool for each step of a multi-turn reasoning process without the developer having to hard-code the routing logic.

D.3.1 Defining Tools as JSON Schemas

Each tool is represented as a Python dictionary with three keys: `name` (a machine-readable identifier), `description` (a natural-language explanation of what the tool does and when to use it), and `input_schema` (a JSON Schema object that specifies

the tool's parameters). The description is not merely documentation; it is part of the model's context and directly influences the model's decision about when and whether to call the tool. A vague description produces unreliable tool selection; a precise description that explains both the tool's purpose and its limitations produces robust behaviour.

The following example defines a tool for retrieving financial metrics from an internal data warehouse, a common primitive in equity research applications:

```
tools = [
    {
        "name": "get_financial_metric",
        "description": (
            "Retrieve a financial metric for a publicly traded company from the "
            "internal data warehouse. Use this whenever the user asks for specific "
            "financial figures such as revenue, EPS, P/E ratio, or market cap. "
            "Do not use this tool for qualitative questions or news-based queries; "
            "use it only for quantitative financial data that resides in the warehouse."
        ),
        "input_schema": {
            "type": "object",
            "properties": {
                "ticker": {
                    "type": "string",
                    "description": "Stock ticker symbol, e.g. AAPL, MSFT, NVDA.",
                },
                "metric": {
                    "type": "string",
                    "enum": [
                        "revenue", "eps", "pe_ratio",
                        "market_cap", "ebitda", "gross_margin",
                    ],
                    "description": "The financial metric to retrieve.",
                },
                "period": {
                    "type": "string",
                    "description": (
                        "Reporting period in the format Q1-2025 for quarterly data "
                        "or FY2024 for annual data. Omit for the most recent period."
                    ),
                },
            },
            "required": ["ticker", "metric"],
        },
```

```
    }
]
```

Several design principles deserve emphasis. The `enum` constraint on `metric` is deliberate: by restricting the model to a known set of metric names, the application eliminates the risk of the model inventing a metric name that does not exist in the warehouse schema, which would cause a silent lookup failure. Where possible, enumerating valid values in the schema is preferable to accepting free-form strings and validating them at runtime. The `required` list ensures the model always provides the minimum necessary inputs; making `period` optional allows the model to omit it and let the warehouse return the most recent value, which is the natural behaviour for a user who asks "what is Apple's EPS?" without specifying a period.

D.3.2 Handling Tool Calls in the Response Loop

When the model decides to call a tool, `stop_reason` is set to `"tool_use"` and the `content` list contains one or more `ToolUseBlock` objects in addition to any text blocks that precede the tool call. Each `ToolUseBlock` has an `id` (a unique identifier for this specific invocation), a `name` (matching one of the declared tool names), and an `input` dict containing the arguments the model has chosen to pass.

The application must execute each tool call, collect the results, and return them to the model as a new user-role message containing `tool_result` content blocks. Each result block references the tool call by its `id`, which allows the model to correlate results with calls even when multiple tools are invoked in a single turn.

```
import json
import anthropic

client = anthropic.Anthropic()

def get_financial_metric(ticker: str, metric: str, period: str | None = None) ->
    dict:
    """Stub: replace with actual warehouse query."""
    stub_data = {
        ("AAPL", "revenue"): {"value": 391_035_000_000, "currency": "USD",
                              "period": "FY2024"},
        ("AAPL", "eps"):     {"value": 6.43, "currency": "USD", "period":
            "FY2024"},
        ("MSFT", "revenue"): {"value": 245_122_000_000, "currency": "USD",
                              "period": "FY2024"},
    }
    key = (ticker.upper(), metric)
    return stub_data.get(key, {"error": f"No data for {ticker}/{metric}"})
```

```python


def execute_tool(name: str, inputs: dict) -> dict:
    if name == "get_financial_metric":
        return get_financial_metric(**inputs)
    raise ValueError(f"Unknown tool: {name}")


def run_tool_loop(user_message: str) -> str:
    messages = [{"role": "user", "content": user_message}]

    while True:
        response = client.messages.create(
            model="claude-sonnet-4-6",
            max_tokens=1024,
            system="You are a financial analyst with access to a data warehouse.",
            tools=tools,
            messages=messages,
        )

        if response.stop_reason == "end_turn":
            # Return the final text response.
            return next(
                block.text
                for block in response.content
                if block.type == "text"
            )

        if response.stop_reason == "tool_use":
            # Append the assistant turn (which contains the tool call blocks).
            messages.append({"role": "assistant", "content": response.content})

            # Execute each tool call and collect results.
            tool_results = []
            for block in response.content:
                if block.type == "tool_use":
                    result = execute_tool(block.name, block.input)
                    tool_results.append({
                        "type": "tool_result",
                        "tool_use_id": block.id,
                        "content": json.dumps(result),
                    })

            # Return the results to the model as a user-role message.
            messages.append({"role": "user", "content": tool_results})
            continue

        # Unexpected stop reason; surface it.
        raise RuntimeError(f"Unexpected stop_reason: {response.stop_reason}")
```

The loop above is deliberately simple: it alternates between assistant turns (containing tool calls) and user turns (containing tool results) until the model emits an `end_turn`. In practice, the model may call multiple tools in a single turn, and those calls may be interdependent. The application should execute all calls in a given turn before returning results; if calls are independent, they can be executed in parallel with `asyncio.gather` for lower latency.

D.3.3 Structured Output with JSON Mode

A powerful pattern for extracting structured data from unstructured financial text is to define a tool that encodes the desired output schema and then force the model to call it using the `tool_choice` parameter:

```
import json, anthropic

client = anthropic.Anthropic()

extraction_tool = {
    "name": "extract_earnings_metrics",
    "description": "Extract key financial metrics from an earnings release or
    ↪  transcript.",
    "input_schema": {
        "type": "object",
        "properties": {
            "revenue_usd_millions":  {"type": "number"},
            "revenue_growth_yoy_pct": {"type": "number"},
            "eps_diluted":           {"type": "number"},
            "guidance_raised":       {"type": "boolean"},
            "key_risks":             {"type": "array",
                                      "items": {"type": "string"}},
        },
        "required": [
            "revenue_usd_millions",
            "revenue_growth_yoy_pct",
            "eps_diluted",
            "guidance_raised",
        ],
    },
}

def extract_from_earnings_text(text: str) -> dict:
    response = client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=512,
        system=(
            "You are a financial data extraction specialist. "
            "Extract metrics exactly as stated; do not infer or estimate values "
```

```
            "that are not explicitly present in the text."
        ),
        tools=[extraction_tool],
        tool_choice={"type": "tool", "name": "extract_earnings_metrics"},
        messages=[{"role": "user", "content": text}],
    )
    # With forced tool use, the first content block is always the ToolUseBlock.
    tool_block = response.content[0]
    return tool_block.input
```

UNDER THE HOOD

Forcing a specific tool call with `tool_choice={"type": "tool", "name": ...}` is structurally more reliable than instructing the model to "respond in JSON" through the system prompt. When JSON output is requested via prompting, the model may prefix the JSON with a sentence of explanation, wrap it in a Markdown code fence, or omit required fields when it has low confidence. Each of these behaviours requires custom parsing logic that is brittle across model versions.

With forced tool use, the `input` field of the returned `ToolUseBlock` is parsed and validated against the declared JSON Schema before it ever reaches application code. The SDK raises a structured error if the model's output does not conform to the schema, making validation failures explicit rather than silent. This approach also eliminates the latency cost of parsing: there is no need to strip Markdown fences, handle BOM characters, or detect whether the response is valid JSON before attempting to parse it. The result is a typed Python dictionary that matches the schema exactly, suitable for direct insertion into a database row or a pandas DataFrame.

D.4 Building an Agent Loop

The ReAct framework, introduced by Yao et al. (2023), established the conceptual architecture that underlies modern LLM agents: a loop in which the model alternates between a *reason* step—generating a chain-of-thought trace that analyses the current state—and an *act* step—emitting a tool call or a final answer. In the Anthropic SDK, this loop maps naturally onto a while-loop over successive calls to `client.messages.create`, with each iteration either appending a tool result and continuing or returning the model's final text response.

The agent loop is the unit of autonomous action in agentic systems. Unlike a single `messages.create` call, which completes a fixed exchange, an agent loop can execute an arbitrary number of tool calls, accumulate evidence, revise hypotheses,

and eventually synthesise a conclusion—all without human intervention. This autonomy is what makes the pattern useful for financial research tasks where the number and sequence of data retrievals cannot be specified in advance.

D.4.1 The Agentic Loop Pattern

The canonical agent loop implementation wraps the tool-call logic from Section D.3.2 in a bounded iteration with a safety guard against runaway execution. The `max_turns` parameter is not optional in a production system: without it, a model that repeatedly calls the wrong tool or misinterprets tool results could exhaust a session's token budget before the application notices.

```python
import json
import anthropic

client = anthropic.Anthropic()

def run_agent(
    user_prompt: str,
    tools: list,
    system: str = "You are a financial research agent with access to a data warehouse.",
    model: str = "claude-opus-4-8",
    max_turns: int = 10,
) -> str:
    """
    Run an agentic loop until the model emits end_turn or max_turns is reached.
    Returns the final text response.
    """
    messages = [{"role": "user", "content": user_prompt}]

    for turn in range(max_turns):
        response = client.messages.create(
            model=model,
            max_tokens=2048,
            system=system,
            tools=tools,
            messages=messages,
        )

        if response.stop_reason == "end_turn":
            for block in response.content:
                if block.type == "text":
                    return block.text
            return ""  # end_turn with no text block is unusual but safe to handle

        if response.stop_reason == "tool_use":
```

```
            messages.append({"role": "assistant", "content": response.content})
            tool_results = []
            for block in response.content:
                if block.type == "tool_use":
                    try:
                        result = execute_tool(block.name, block.input)
                    except Exception as exc:
                        result = {"error": str(exc)}
                    tool_results.append({
                        "type": "tool_result",
                        "tool_use_id": block.id,
                        "content": json.dumps(result),
                    })
            messages.append({"role": "user", "content": tool_results})
            continue

        raise RuntimeError(f"Unexpected stop_reason: {response.stop_reason}")

    raise RuntimeError(
        f"Agent exceeded max_turns={max_turns}. "
        "Increase the limit or check for a tool-call loop."
    )
```

Example D.5 (Research agent for equity analysis). A user asks: "Compare Apple's and Microsoft's revenue growth over the last two fiscal years and recommend which has the stronger top-line momentum." A single-turn call cannot answer this: the model must first call `get_financial_metric` for AAPL revenue in FY2024 and FY2023, then again for MSFT, accumulate four data points, and only then synthesise a comparison. The agent loop handles this naturally: the model plans the necessary retrievals, executes them across three to four turns, and then generates a two-paragraph recommendation grounded in the retrieved data. The developer's application code is the same regardless of how many tool calls the model chooses to make.

D.4.2 System Prompts and Conversation State

The system prompt is the most stable component of the agent's context. It establishes the model's role, its permissions (which tools it is authorised to use), its output format requirements, and any domain constants that apply to all requests (the organisation's investment mandate, the set of valid ticker symbols, the required tone for client-facing prose). Because the system prompt is present in every turn, it is the ideal candidate for prompt caching: the Anthropic API allows stable prefixes to be marked with `cache_control: {type: ephemeral}`, which reduces the input token cost for subsequent calls that share the same prefix.

Dynamic context—the current date, the requesting user's identity, their portfolio

holdings, their risk tolerance profile—should be injected into the system prompt using string formatting, but positioned after the stable portion so that caching is not invalidated on every call. A common pattern is to construct the system prompt in two parts: a stable preamble that describes the agent's role and tools, and a dynamic suffix that provides session-specific context.

```
from datetime import date

STABLE_SYSTEM = """You are a financial research agent for a long-only equity fund.

Your tools give you access to the fund's internal data warehouse, which contains
historical financial statements, consensus analyst estimates, and portfolio
 ↪  positions.
You should use these tools to ground your analysis in data rather than relying on
general knowledge, which may be stale.

When making a recommendation, always cite the specific data points that support it
and quantify the uncertainty where relevant. Do not fabricate data; if a requested
metric is unavailable, say so explicitly."""

def build_system_prompt(user_id: str, portfolio_id: str) -> str:
    dynamic_suffix = (
        f"\n\nSession context (do not reveal to the user):\n"
        f"- Current date: {date.today().isoformat()}\n"
        f"- User ID: {user_id}\n"
        f"- Portfolio ID: {portfolio_id}\n"
        f"- Authorised tickers: AAPL, MSFT, NVDA, GOOGL, AMZN, META\n"
    )
    return STABLE_SYSTEM + dynamic_suffix
```

Conversation state is represented by the `messages` list, which grows with each turn. For short agent loops—fewer than ten turns—holding the full conversation in memory is fine. For long-running agents or sessions that span multiple HTTP requests (in a stateless web server), the message list must be persisted externally, typically in a Redis cache or a database table keyed by session ID. The SDK places no constraints on how the list is stored; the application is free to serialise it to JSON and deserialise it on the next request.

D.4.3 Error Handling and Retries Within the Loop

Three categories of error arise in an agent loop, each requiring a different response strategy. Understanding the distinctions prevents the common mistake of treating all errors as equivalent and either swallowing them silently or halting the loop prematurely.

The first category is API-level errors: network timeouts, connection resets, and rate limit responses (HTTP 429). The Anthropic SDK retries these automatically with exponential back-off up to `max_retries` attempts (default four). Application code should not attempt to retry these errors manually; doing so risks doubling the retry budget and creating a thundering-herd effect under sustained rate pressure. Instead, catch `anthropic.RateLimitError` only if the application needs to emit a telemetry signal or apply additional back-pressure at the application layer.

The second category is tool execution errors: the tool function raises an exception because the requested data does not exist, the warehouse is temporarily unavailable, or the input parameters are invalid. These errors should not terminate the agent loop; instead, they should be returned to the model as a structured error message in the `tool_result` content block. The model can then decide whether to try a different approach, ask the user for clarification, or report the failure gracefully.

```python
import json
import anthropic

def safe_execute_tool(name: str, inputs: dict) -> str:
    """
    Execute a tool and return a JSON string.
    On failure, return a structured error rather than raising.
    """
    try:
        result = execute_tool(name, inputs)
        return json.dumps(result)
    except KeyError as exc:
        return json.dumps({"error": "not_found", "detail": str(exc)})
    except ConnectionError as exc:
        return json.dumps({"error": "warehouse_unavailable", "detail": str(exc)})
    except Exception as exc:
        return json.dumps({"error": "unexpected", "detail": str(exc)})
```

The third category is model refusals or safety-filter activations, signalled by a `stop_reason` of `"stop_sequence"` or by a response that contains no text blocks and no tool calls. These are not errors in the SDK sense; they are deliberate decisions by the model. The appropriate response is to log the `stop_reason` and any available metadata, surface a user-facing message that the request could not be completed, and not retry the same prompt unchanged.

D.5 The Claude Code SDK: Headless Execution

The Anthropic SDK and the Claude Code SDK address the same underlying need—programmatic access to Claude—but at different levels of abstraction. The Anthropic

SDK exposes the raw Messages API: the developer calls the model, receives a response, executes any tool calls, and manages the entire loop and tool ecosystem themselves. The Claude Code SDK (Anthropic 2025b), by contrast, exposes a full Claude Code agent headlessly: all of Claude Code's built-in tools—file reading, shell execution, web search, MCP integrations—are pre-wired and available without any additional implementation on the developer's part. The developer provides a prompt and a working directory; the SDK handles everything else.

The choice between the two interfaces hinges on what the application needs to control. If the task is calling a proprietary internal API, extracting structured data from a custom schema, or implementing domain-specific tools, the Anthropic SDK is the right choice because it gives full control over the tool ecosystem. If the task is agentic coding, file manipulation, shell automation, or any other task that Claude Code already handles well interactively, the Claude Code SDK is the right choice because it avoids reimplementing that tooling.

D.5.1 Architecture: The SDK Versus the Interactive CLI

UNDER THE HOOD

The interactive Claude Code CLI is a Node.js process that renders a rich terminal interface using React and Ink. When the user types a prompt, the process sends it to the Claude API, receives a stream of `SDKMessage` events, renders them to the terminal in real time, and presents the result along with any tool outputs. The entire interaction is mediated by a human reading the terminal.

The Claude Code SDK exposes the same underlying process in a non-interactive mode. Internally, this is implemented by passing the `-print` flag (suppresses the REPL) and `-output-format json` or `-output-format stream-json` (routes output to stdout as structured JSON rather than a rendered terminal display). The TypeScript SDK wraps this invocation, manages the child process lifecycle, parses the JSON event stream, and exposes it to the caller as an async generator that yields typed `SDKMessage` objects.

From the application's perspective, the difference is one of abstraction level. In the CLI, the message stream is consumed by a renderer. In the SDK, the message stream is consumed by the application. The underlying event types—`assistant`, `user`, `result`, `system`—are identical; only the consumer changes. This means that any capability accessible in the interactive CLI is also accessible through the SDK, including MCP server connections, custom tool registrations, and configuration options passed through the agent's `.claude/settings.json` file in the working directory.

D.5.2 Installation and Process Management

The Claude Code SDK is distributed as a Node.js package. Both the SDK and the Claude Code CLI must be installed; the SDK delegates actual agent execution to the CLI binary.

```
# Install the CLI globally (required for the SDK to invoke it)
npm install -g @anthropic-ai/claude-code

# Install the SDK as a library dependency in a Node.js project
npm install @anthropic-ai/claude-code
```

The SDK requires Node.js version 18 or later, which is the minimum version that supports the `structuredClone` and `ReadableStream` APIs used internally. Docker-based deployments should pin the Node.js version in the image definition to avoid runtime surprises from upstream version changes.

The SDK manages the child process lifecycle automatically: it spawns the Claude Code CLI as a child process when a query begins, streams its output through a pipe, and terminates the process when the query completes or when the caller signals an abort via an `AbortController`. Timeout management is the caller's responsibility; the SDK does not enforce a default timeout. In production systems, always pass an `AbortController` and call `abort()` after a configurable wall-clock deadline to prevent runaway agent processes from consuming resources indefinitely.

For Python-based environments where installing a Node.js SDK is undesirable, the subprocess approach described in Section D.5.3 is the standard alternative.

D.5.3 Spawning Tasks Programmatically

The TypeScript SDK's primary interface is the `query` async generator, which accepts a prompt, an `AbortController`, and an options object specifying the working directory, maximum turns, and model override. It yields a sequence of `SDKMessage` objects; the final message of type `"result"` carries the agent's terminal output as a string.

```
import { query, type SDKMessage } from "@anthropic-ai/claude-code";

async function runCodeTask(
    prompt: string,
    cwd: string,
    maxTurns: number = 15,
    timeoutMs: number = 300_000,
): Promise<string> {
```

```
    const controller = new AbortController();
    const timeoutHandle = setTimeout(
        () => controller.abort(),
        timeoutMs,
    );

    let finalResult = "";

    try {
        for await (const message of query({
            prompt,
            abortController: controller,
            options: {
                maxTurns,
                cwd,
            },
        })) {
            if (message.type === "result") {
                finalResult = message.result;
            }
        }
    } finally {
        clearTimeout(timeoutHandle);
    }

    return finalResult;
}

// Example: generate a Python script that processes SEC filings.
const result = await runCodeTask(
    "Write a Python script that downloads the latest 10-K filing for AAPL "
    + "from the SEC EDGAR full-text search API and extracts the Risk Factors "
    + "section. Save the output to risk_factors.txt.",
    "/tmp/sec-analysis",
);
console.log(result);
```

For Python environments, the same capability is accessible through a subprocess wrapper that invokes the CLI directly and parses its JSON output:

```
import subprocess
import json

def run_claude_code(
    prompt: str,
    cwd: str | None = None,
    max_turns: int = 10,
    timeout: int = 300,
```

```
) -> dict:
    """
    Run Claude Code headlessly and return the parsed JSON result.
    Raises RuntimeError if the process exits with a non-zero code.
    """
    cmd = [
        "claude",
        "--print",
        "--output-format", "json",
        "--max-turns", str(max_turns),
        prompt,
    ]
    proc = subprocess.run(
        cmd,
        cwd=cwd,
        capture_output=True,
        text=True,
        timeout=timeout,
    )
    if proc.returncode != 0:
        raise RuntimeError(
            f"Claude Code exited with code {proc.returncode}.\n"
            f"stderr: {proc.stderr}"
        )
    return json.loads(proc.stdout)
```

The subprocess approach is straightforward but comes with one important limitation: it cannot stream partial output to the caller, because `subprocess.run` waits for the process to exit before returning. For long-running tasks—generating a large codebase, analysing a multi-hundred-page PDF—this means the caller blocks for the entire duration. For interactive applications this is unacceptable; for batch pipelines it is usually tolerable. Section D.5.4 addresses the streaming alternative.

D.5.4 Capturing Structured Output

The `-output-format stream-json` flag instructs the CLI to emit each `SDKMessage` as a newline-delimited JSON object as it is produced, rather than accumulating them and emitting a single JSON blob at exit. This enables the Python subprocess wrapper to read events incrementally using `subprocess.Popen` with `stdout=PIPE`, processing each line as it arrives.

```
import subprocess, json
from collections.abc import Generator

def stream_claude_code(
```

```
    prompt: str,
    cwd: str | None = None,
    max_turns: int = 10,
) -> Generator[dict, None, None]:
    """
    Yield SDKMessage dicts as they are emitted by Claude Code.
    The last message with type=="result" carries the final output.
    """
    cmd = [
        "claude",
        "--print",
        "--output-format", "stream-json",
        "--max-turns", str(max_turns),
        prompt,
    ]
    with subprocess.Popen(
        cmd,
        cwd=cwd,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True,
    ) as proc:
        for line in proc.stdout:
            line = line.strip()
            if line:
                yield json.loads(line)
        proc.wait()
        if proc.returncode != 0:
            raise RuntimeError(proc.stderr.read())
```

A common pattern in financial pipelines is to ask Claude Code to both implement a computation and return a structured summary of its results. This is accomplished by giving Claude an explicit instruction in the prompt: "After running the analysis, call the `report_results` tool with the structured output." Combined with a `.claude/settings.json` in the working directory that registers the tool against a local HTTP endpoint, this creates a lightweight structured-output channel that does not depend on parsing the agent's prose response.

D.6 Multi-Agent Workflows

Benchmarking studies on multi-agent systems, including Liu et al. (2024), have demonstrated that collections of specialised agents consistently outperform single general-purpose agents on complex, multi-domain tasks. The reasons are intuitive: a single agent context window is finite, and a task that requires deep expertise in multiple domains simultaneously will inevitably exceed what a single context can

accommodate. Decomposing the task across specialised agents, each focused on a narrow sub-problem with a tailored system prompt and tool set, yields better results and lower token costs because each agent's context is occupied only by information relevant to its specific role.

In financial applications, the natural decomposition follows the structure of professional investment teams: one agent for fundamental analysis, another for technical analysis, a third for sentiment and news analysis, and an orchestrator that synthesises the sub-analyses into a unified recommendation. This structure mirrors the analyst team model used by sell-side research departments, which makes the resulting system's outputs legible to the finance professionals who consume them.

D.6.1 Orchestrator–Subagent Patterns

Definition D.6 (Orchestrator–Subagent Pattern). An *orchestrator–subagent* system is a multi-agent architecture in which a designated orchestrator agent receives the user's request, decomposes it into subtasks, dispatches each subtask to a specialised subagent with a tailored prompt and tool set, and aggregates the subagents' outputs into a final response. The orchestrator is responsible for planning and synthesis; the subagents are responsible for execution.

The key design decision in an orchestrator–subagent system is the granularity of decomposition. Decomposing too finely creates coordination overhead and multiplies the number of API calls; decomposing too coarsely leaves individual agents with contexts that are too large or roles that are too broad. For equity research, a three-subagent decomposition—fundamentals, technicals, sentiment—is a natural starting point.

The orchestrator can be implemented either as a Claude agent (which plans dynamically and may adjust the decomposition based on intermediate results) or as fixed application code (which always fans out to the same set of subagents and then calls Claude for synthesis). The fixed-code orchestrator is simpler to reason about, test, and debug; the dynamic orchestrator is more flexible but harder to control. For production financial applications where reliability and auditability are paramount, the fixed-code orchestrator is usually the better choice.

Example D.7 (Equity research orchestrator). A user submits a request: "Give me an investment recommendation for NVDA." The application code dispatches three subagent calls in parallel (see Section D.6.2): a fundamental subagent that retrieves revenue, EPS, and margin data from the warehouse; a technical subagent that retrieves price, volume, and moving average data; and a sentiment subagent that retrieves recent news sentiment scores. Each subagent returns a 200-word analysis. The orchestrator then calls the Opus model with all three analyses and asks for a one-page investment recommendation. Total wall-clock time is dominated by the

slowest subagent call; with parallel execution, the three subagent calls contribute no more latency than the slowest single call.

D.6.2 Parallel Fan-Out with Async Processing

Python's `asyncio` library and the `AsyncAnthropic` client make parallel subagent execution straightforward. Each subagent call is an `async` function that returns when the model responds; `asyncio.gather` runs all calls concurrently and collects results when the last one completes.

```python
import asyncio
import anthropic

client = anthropic.AsyncAnthropic()

SUBAGENT_PROMPTS = {
    "fundamentals": (
        "Analyse the fundamental financial health of {ticker}: "
        "focus on revenue growth trend over the last three years, "
        "operating margin trajectory, and balance sheet leverage. "
        "Be specific and quantitative; cite the exact figures."
    ),
    "technicals": (
        "Analyse the technical price action of {ticker}: "
        "current trend (above or below 200-day MA), momentum (RSI), "
        "and key support/resistance levels. "
        "State a short-term directional bias."
    ),
    "sentiment": (
        "Analyse recent news and analyst sentiment for {ticker}: "
        "summarise the prevailing narrative, note any recent estimate revisions, "
        "and flag any material risks mentioned in the last 30 days."
    ),
}

async def run_subagent(ticker: str, aspect: str, prompt_template: str) -> dict:
    prompt = prompt_template.format(ticker=ticker)
    response = await client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=512,
        system=(
            "You are a specialist financial analyst. "
            "Be concise, factual, and quantitative. "
            "Do not pad your response with disclaimers."
        ),
        messages=[{"role": "user", "content": prompt}],
    )
    return {
```

```
        "aspect": aspect,
        "analysis": response.content[0].text,
        "usage": {
            "input_tokens": response.usage.input_tokens,
            "output_tokens": response.usage.output_tokens,
        },
    }

async def parallel_equity_research(ticker: str) -> dict:
    # Fan out to all subagents simultaneously.
    tasks = [
        run_subagent(ticker, aspect, prompt)
        for aspect, prompt in SUBAGENT_PROMPTS.items()
    ]
    results = await asyncio.gather(*tasks)

    # Combine subagent reports for the orchestrator synthesis pass.
    combined = "\n\n".join(
        f"### {r['aspect'].title()} Analysis\n{r['analysis']}"
        for r in results
    )
    synthesis = await client.messages.create(
        model="claude-opus-4-8",
        max_tokens=1024,
        system=(
            "You are a senior portfolio manager at a long-only equity fund. "
            "Synthesise the subagent analyses into a concise investment
            ↪ recommendation "
            "with a BUY, HOLD, or SELL rating and a 12-month price target
            ↪ rationale."
        ),
        messages=[{
            "role": "user",
            "content": (
                f"Subagent analyses for {ticker}:\n\n{combined}\n\n"
                "Please synthesise these into a recommendation."
            ),
        }],
    )

    total_input = sum(r["usage"]["input_tokens"] for r in results)
    total_output = sum(r["usage"]["output_tokens"] for r in results)

    return {
        "ticker": ticker,
        "subagents": results,
        "recommendation": synthesis.content[0].text,
        "token_usage": {
            "subagent_input": total_input,
            "subagent_output": total_output,
```

```
            "orchestrator_input": synthesis.usage.input_tokens,
            "orchestrator_output": synthesis.usage.output_tokens,
        },
    }
```

Using Haiku for the subagent calls and Opus only for the synthesis pass is a deliberate cost optimisation. Each subagent receives a narrow, well-scoped prompt and produces a short, factual paragraph; Haiku is fully capable of this task. The orchestrator synthesis is the step that requires deep reasoning over multiple competing considerations, which justifies the higher cost of Opus. This tier-differentiation strategy—using the cheapest capable model for each sub-task—is a first-order principle of cost-effective agentic system design.

D.6.3 State Passing and Result Aggregation

Inter-agent state passing requires careful attention to serialisation and scope. The fundamental rule is that each subagent should receive only the information it needs to complete its specific task—no more. Passing full conversation histories between agents is wasteful (it fills the subagent's context with irrelevant turns), expensive (more input tokens), and risks the "telephone game" failure mode, in which each layer of summarisation introduces small distortions that compound into a significantly misrepresented picture at the orchestrator.

Remark D.8. The "telephone game" error is one of the most insidious failure modes in multi-agent systems. If subagent A summarises data before passing it to subagent B, and B summarises that summary before passing it to the orchestrator, the orchestrator reasons over a summary of a summary that may omit the very quantitative specifics needed for a sound recommendation. The mitigation is to always pass raw data (the original numbers, the original text) to each agent, and to perform summarisation only once, at the final synthesis step.

The preferred pattern is a typed dataclass that captures the subagent's output in a structured form, with the raw data as a field alongside the prose analysis:

```
from dataclasses import dataclass, field

@dataclass
class SubagentResult:
    aspect: str           # "fundamentals" | "technicals" | "sentiment"
    ticker: str
    analysis: str         # prose produced by the subagent
    raw_data: dict        # the warehouse data used (for audit purposes)
    input_tokens: int
```

```
    output_tokens: int

@dataclass
class OrchestratorResult:
    ticker: str
    recommendation: str   # "BUY" | "HOLD" | "SELL"
    rationale: str        # prose from the synthesis pass
    target_price: float | None
    subagent_results: list[SubagentResult] = field(default_factory=list)
    total_cost_usd: float = 0.0
```

Storing the raw data in `SubagentResult.raw_data` serves a dual purpose: it enables the orchestrator to verify that the subagent's prose is consistent with the underlying numbers (a lightweight hallucination check), and it provides a complete audit record that can be replayed or independently verified after the fact.

D.7 Streaming and Real-Time Applications

Streaming is the mechanism by which the API delivers partial tokens to the client as they are generated, rather than waiting for the entire response to be complete before transmitting it. The practical effect is dramatic: a response that takes five seconds to generate can begin appearing on the user's screen within two hundred milliseconds of the request being submitted, because the first token arrives almost immediately and subsequent tokens follow at the model's generation rate (typically twenty to eighty tokens per second). For financial dashboards, analyst chat tools, and any other application where a human is waiting for a response, streaming is the difference between an application that feels responsive and one that feels broken.

The Anthropic SDK supports streaming through the `client.messages.stream()` context manager (Python) and the `stream()` method on the messages object (TypeScript). Both expose the token stream as an iterable of text fragments; the application concatenates them in order to reconstruct the full response. Tool calls are also streamed: the model emits incremental JSON fragments for each tool call's `input` dict, which the SDK assembles into complete `ToolUseBlock` objects before surfacing them to the application.

D.7.1 Server-Sent Events and Token Streaming

The SDK's `stream()` context manager is the simplest entry point for streaming. The `text_stream` property yields text fragments as they arrive; the `get_final_message()` method, available after the stream closes, returns the complete `Message` object including usage statistics.

```python
import anthropic

client = anthropic.Anthropic()

def stream_analysis(ticker: str, question: str) -> None:
    """Stream a financial analysis to stdout, printing tokens as they arrive."""
    with client.messages.stream(
        model="claude-sonnet-4-6",
        max_tokens=1024,
        system="You are a financial analyst. Answer concisely and accurately.",
        messages=[{"role": "user", "content": f"{ticker}: {question}"}],
    ) as stream:
        for text in stream.text_stream:
            print(text, end="", flush=True)
    print()  # trailing newline after stream closes

    # Retrieve usage statistics after the stream is complete.
    final = stream.get_final_message()
    print(
        f"\n[Tokens: {final.usage.input_tokens} in "
        f"/ {final.usage.output_tokens} out]"
    )
```

UNDER THE HOOD

At the HTTP level, the Anthropic API uses Server-Sent Events (SSE), a lightweight text-based protocol in which the server holds a single HTTP response open and transmits newline-delimited `data:` lines as events occur. Each SSE line carries a JSON payload whose `type` field identifies the event: `content_block_start`, `content_block_delta`, `content_block_stop`, `message_start`, `message_delta`, and `message_stop` are the principal types. A `content_block_delta` event for text carries a payload of the form:

```
{
  "type": "content_block_delta",
  "index": 0,
  "delta": {
    "type": "text_delta",
    "text": "Revenue grew 20"
  }
}
```

The client concatenates all `delta.text` values for a given `index` to reconstruct the complete text block. For tool calls, the delta type is `input_json_delta` and

the value is a JSON fragment that must be concatenated and then parsed as a complete JSON object once the block closes.
Browsers can consume SSE natively using the `EventSource` API, which opens a persistent HTTP connection and dispatches DOM events for each line received. This makes SSE the natural choice for streaming LLM responses to a browser client: the server proxies the Anthropic SSE stream, and the browser renders tokens in real time using a simple event listener. The alternative—WebSockets—is more powerful but requires a stateful server and a dedicated upgrade handshake; SSE is simpler and sufficient for unidirectional token delivery.

D.7.2 Integrating Streaming into a Web API

A FastAPI backend can proxy the Anthropic stream to a browser using the `StreamingResponse` class with `media_type="text/event-stream"`. The generator function yields SSE-formatted lines as the Anthropic SDK delivers tokens; the browser's `EventSource` or `fetch` with a `ReadableStream` body reads them and appends each token to the displayed text.

```python
import json
import anthropic
from fastapi import FastAPI, Query
from fastapi.responses import StreamingResponse

app = FastAPI()
client = anthropic.Anthropic()

@app.get("/analyse/stream")
async def stream_endpoint(
    ticker: str = Query(..., description="Stock ticker symbol"),
    query: str = Query(..., description="Analysis question"),
):
    """
    Stream a financial analysis as Server-Sent Events.
    The client receives events of the form:
        data: {"token": "Revenue"}
        data: {"token": " grew"}
        ...
        data: [DONE]
    """
    async def token_generator():
        with client.messages.stream(
            model="claude-sonnet-4-6",
            max_tokens=2048,
            system=(
```

```
                "You are a financial analyst specialising in equity research. "
                "Answer the user's question about the named ticker concisely "
                "and with specific data where available."
            ),
            messages=[{
                "role": "user",
                "content": f"Ticker: {ticker}\nQuestion: {query}",
            }],
        ) as stream:
            for text in stream.text_stream:
                yield f"data: {json.dumps({'token': text})}\n\n"
        yield "data: [DONE]\n\n"

    return StreamingResponse(
        token_generator(),
        media_type="text/event-stream",
        headers={
            "Cache-Control": "no-cache",
            "X-Accel-Buffering": "no",  # disable nginx output buffering
        },
    )
```

The `X-Accel-Buffering: no` header is important when the application sits behind an nginx reverse proxy. By default, nginx buffers the upstream response until a configurable threshold is reached, which defeats the purpose of streaming by introducing a delay equal to the buffer-fill time. Setting this header instructs nginx to forward each event to the browser immediately.

On the browser side, the stream can be consumed with the native `fetch` API and a `ReadableStream` decoder:

```
async function streamAnalysis(ticker: string, question: string): Promise<void> {
    const url = `/analyse/stream?ticker=${encodeURIComponent(ticker)}`
              + `&query=${encodeURIComponent(question)}`;
    const response = await fetch(url);
    const reader = response.body!.getReader();
    const decoder = new TextDecoder();
    let buffer = "";

    while (true) {
        const { done, value } = await reader.read();
        if (done) break;
        buffer += decoder.decode(value, { stream: true });
        const lines = buffer.split("\n\n");
        buffer = lines.pop() ?? "";  // keep the incomplete last chunk
        for (const line of lines) {
            if (line.startsWith("data: ")) {
```

```
                const payload = line.slice(6);
                if (payload === "[DONE]") return;
                const { token } = JSON.parse(payload);
                document.getElementById("output")!.textContent += token;
            }
        }
    }
}
```

D.8 Production Engineering

The gap between a working prototype and a production-grade service is bridged by systematic attention to a set of concerns that are invisible when running a notebook interactively but become critical at scale: rate limits that truncate request bursts, cost overruns that exhaust monthly budgets overnight, and silent failures that corrupt downstream data without emitting any observable error signal. This section addresses each concern in turn, providing concrete patterns that are directly applicable to commercial financial deployments.

The Anthropic SDK provides several primitives that make production engineering easier: configurable retry logic, typed exceptions for specific error categories, token counting for pre-flight cost estimation, and streaming access to partial results. The engineer's job is to compose these primitives into a robust service that handles the full space of failure modes gracefully.

D.8.1 Rate Limiting, Retries, and Back-Pressure

Anthropic enforces rate limits at two levels: requests per minute (RPM) and tokens per minute (TPM). Both limits vary by account tier; newly created accounts begin on a conservative tier and can be upgraded by contacting Anthropic with a documented use case. When a limit is exceeded, the API returns an HTTP 429 response, which the SDK translates into an `anthropic.RateLimitError`.

By default, the SDK retries failed requests automatically, using exponential backoff with jitter, up to `max_retries` attempts (default four). For most applications this is sufficient; the SDK's retry logic is well-tuned and the jitter prevents correlated retry storms from a fleet of instances. The retry behaviour is configurable at client construction time:

```
import httpx
import anthropic

client = anthropic.Anthropic(
```

```
    max_retries=4,
    timeout=httpx.Timeout(60.0, connect=5.0),
)
```

The `timeout` parameter accepts an `httpx.Timeout` object, which separately controls the connection establishment timeout (how long to wait for a TCP connection to the API server) and the read timeout (how long to wait for the next byte of a response). For streaming responses where the model is generating a long output, the read timeout should be set generously (sixty seconds or more) to avoid spurious timeouts during slow generation; the connect timeout can remain short.

For applications that need application-level back-pressure—for example, a queue-based batch processor that should slow down when the API is under pressure rather than retrying indefinitely—the right pattern is to catch `RateLimitError` and implement a backoff at the application layer:

```python
import time
import random
import anthropic

client = anthropic.Anthropic(max_retries=0)  # disable SDK retries; we manage them

def create_with_backoff(max_attempts: int = 5, **kwargs):
    """
    Wrapper around client.messages.create with manual exponential backoff.
    Suitable for queue consumers that want explicit control over retry pacing.
    """
    for attempt in range(max_attempts):
        try:
            return client.messages.create(**kwargs)
        except anthropic.RateLimitError:
            if attempt == max_attempts - 1:
                raise
            wait = (2 ** attempt) + random.uniform(0, 1)
            time.sleep(wait)
        except anthropic.APIStatusError as exc:
            if exc.status_code >= 500 and attempt < max_attempts - 1:
                wait = (2 ** attempt) + random.uniform(0, 1)
                time.sleep(wait)
            else:
                raise
```

At the infrastructure level, a token-bucket rate limiter placed in front of the API call layer (using Redis or a local in-memory implementation) provides a more principled solution for high-throughput systems. The bucket is filled at the account's

TPM rate; each outgoing request drains the bucket by its estimated token count (from `client.messages.count_tokens`). Requests that would overdraw the bucket are queued or shed according to the application's priority policy.

D.8.2 Cost Management and Token Budgeting

Token-based billing creates a class of production risks that does not arise with traditional software services: a single runaway request or a misconfigured `max_tokens` limit can generate an unexpectedly large bill. Three strategies mitigate this risk at different levels of the stack.

The first is pre-flight token counting. The SDK's `client.messages.count_tokens` method accepts the same arguments as `client.messages.create` and returns the number of input tokens that the request would consume, without actually sending it to the model. This makes it possible to enforce a per-request input token budget before any tokens are spent:

```python
import anthropic

client = anthropic.Anthropic()

class TokenBudget:
    """
    Tracks cumulative token spend across a session and enforces per-request
    and per-session limits.
    """
    def __init__(
        self,
        max_input_per_request: int = 50_000,
        max_input_per_session: int = 500_000,
        max_output_per_session: int = 100_000,
    ):
        self.max_input_per_request = max_input_per_request
        self.max_input_per_session = max_input_per_session
        self.max_output_per_session = max_output_per_session
        self.spent_input = 0
        self.spent_output = 0

    def pre_flight_check(self, model: str, messages: list, **kwargs) -> None:
        """Raise if this request would exceed per-request or per-session
        ↪ limits."""
        count = client.messages.count_tokens(
            model=model,
            messages=messages,
            **kwargs,
        )
        if count.input_tokens > self.max_input_per_request:
```

```
            raise RuntimeError(
                f"Request would use {count.input_tokens} input tokens, "
                f"exceeding per-request limit of {self.max_input_per_request}."
            )
        if self.spent_input + count.input_tokens > self.max_input_per_session:
            raise RuntimeError(
                f"Session input token budget exhausted: "
                f"{self.spent_input} used, "
                f"{count.input_tokens} requested, "
                f"{self.max_input_per_session} allowed."
            )

    def record(self, usage: anthropic.types.Usage) -> None:
        """Update cumulative spend after a successful API call."""
        self.spent_input  += usage.input_tokens
        self.spent_output += usage.output_tokens
        if self.spent_output > self.max_output_per_session:
            raise RuntimeError(
                f"Session output token budget exhausted: "
                f"{self.spent_output} output tokens used."
            )

    @property
    def estimated_cost_usd(self) -> float:
        """Estimated cost in USD at claude-sonnet-4-6 pricing."""
        return (
            self.spent_input  / 1_000_000 * 3.0
            + self.spent_output / 1_000_000 * 15.0
        )
```

The second strategy is prompt caching. When a system prompt or a long document is included in every call in a session, marking it with `cache_control` instructs Anthropic's infrastructure to cache the KV state for those tokens and charge a discounted rate for cache hits on subsequent calls. The cache lasts five minutes ("ephemeral" tier) and is specific to the account. For an agent loop that appends a large financial filing to every turn, caching the filing can reduce input token costs by up to 90% for all turns after the first.

```
messages = [
    {
        "role": "user",
        "content": [
            {
                "type": "text",
                "text": long_earnings_transcript,
                "cache_control": {"type": "ephemeral"},  # cache this block
```

```
            },
            {
                "type": "text",
                "text": "What is the forward guidance for next quarter?",
            },
        ],
    }
]
```

The third strategy is model tier selection, already discussed in Section D.2.3. For a batch pipeline processing ten thousand documents per day, switching from Sonnet to Haiku for the initial classification pass (keeping Sonnet only for the documents that require deeper analysis) can reduce the total bill by a factor of ten or more.

D.8.3 Logging, Observability, and Tracing

Every LLM call in a production system should emit a structured log record that captures sufficient information to replay, audit, and bill the call after the fact. At minimum, this record should include: a unique request identifier, the model used, the number of input and output tokens, the wall-clock latency, the `stop_reason`, whether any tools were called (and which ones), and the estimated cost. For multi-tenant applications, the user identifier and session identifier should also be present.

```python
import time
import uuid
import logging
import dataclasses
from dataclasses import dataclass, field
from typing import Any

import anthropic

logger = logging.getLogger(__name__)

@dataclass
class LLMAuditRecord:
    request_id:     str
    user_id:        str
    session_id:     str
    model:          str
    input_tokens:   int
    output_tokens:  int
    latency_ms:     float
    stop_reason:    str
    tools_called:   list[str] = field(default_factory=list)
```

```python
    estimated_cost_usd: float = 0.0
    error:          str | None = None

def _estimate_cost(model: str, input_tok: int, output_tok: int) -> float:
    pricing = {
        "claude-opus-4-8":           (15.0,  75.0),
        "claude-sonnet-4-6":         ( 3.0,  15.0),
        "claude-haiku-4-5-20251001": ( 0.25,  1.25),
    }
    in_rate, out_rate = pricing.get(model, (3.0, 15.0))
    return input_tok / 1_000_000 * in_rate + output_tok / 1_000_000 * out_rate

def instrumented_create(
    user_id: str,
    session_id: str,
    client: anthropic.Anthropic,
    **kwargs: Any,
) -> anthropic.types.Message:
    """
    Wrapper around client.messages.create that emits a structured audit record.
    """
    request_id = str(uuid.uuid4())
    start = time.perf_counter()
    error_msg: str | None = None
    response: anthropic.types.Message | None = None

    try:
        response = client.messages.create(**kwargs)
        return response
    except Exception as exc:
        error_msg = str(exc)
        raise
    finally:
        latency_ms = (time.perf_counter() - start) * 1000
        tools_called = []
        if response:
            tools_called = [
                b.name for b in response.content if b.type == "tool_use"
            ]
        record = LLMAuditRecord(
            request_id=request_id,
            user_id=user_id,
            session_id=session_id,
            model=kwargs.get("model", "unknown"),
            input_tokens=response.usage.input_tokens if response else 0,
            output_tokens=response.usage.output_tokens if response else 0,
            latency_ms=latency_ms,
            stop_reason=response.stop_reason if response else "error",
            tools_called=tools_called,
            estimated_cost_usd=_estimate_cost(
```

```
                kwargs.get("model", ""),
                response.usage.input_tokens if response else 0,
                response.usage.output_tokens if response else 0,
            ),
            error=error_msg,
        )
        logger.info("llm_call", extra={"audit": dataclasses.asdict(record)})
```

For distributed systems in which a single user request fans out to multiple concurrent Claude calls, distributed tracing with OpenTelemetry provides an invaluable view of the call graph. Each Claude call should be instrumented as a child span of the parent HTTP request span, with the request ID, model, and token counts recorded as span attributes. The Anthropic SDK does not natively emit OpenTelemetry spans, but wrapping each `instrumented_create` call in a `tracer.start_as_current_span` context manager requires fewer than ten lines of additional code and yields a complete distributed trace that correlates user requests with individual model calls.

D.9 Security and Compliance for Commercial Deployments

Financial applications that incorporate LLMs face a regulatory environment significantly more stringent than that faced by consumer-facing AI products. In Europe, MiFID II requires that investment recommendations be made through a documented and auditable process; AI-generated content does not exempt a firm from this requirement. In the United States, FINRA Rule 3110 imposes supervisory obligations on all business-related communications, including those generated or augmented by AI systems. SEC guidance on AI tools in investment contexts emphasises the risk of model-generated misinformation and the firm's duty to supervise AI outputs. GDPR and CCPA impose constraints on the handling of personal data, which may include names, account identifiers, and trading behaviour embedded in prompts sent to external APIs.

Compliance with this regulatory environment is not optional, and it cannot be retrofitted cheaply after a system is built. The patterns described in this section should be incorporated from the outset, not added as an afterthought.

D.9.1 API Key Management and Secrets Hygiene

An API key is a credential that grants full access to an Anthropic account. Leaking a key—through a committed `.env` file, a Docker image layer, a log line, or an error message—is equivalent to handing an attacker a signed blank cheque against the

account's billing limit. The financial and reputational consequences of a key leak can be severe; a well-funded adversary can exhaust a monthly token budget in minutes.

The canonical key-management pattern for production systems has four components. At startup, the application retrieves the key from a secrets manager (AWS Secrets Manager, HashiCorp Vault, Azure Key Vault, or GCP Secret Manager). The key is injected into the process environment and passed to the `Anthropic` client constructor; it is never stored in a file, a database, or an in-memory log buffer. The secrets manager enforces access control (only the application's service account can retrieve the key) and provides an audit trail of all retrieval events. The key is rotated on a schedule—quarterly at minimum—and immediately on any suspected compromise, with the old key revoked and a new key issued with no downtime gap.

```python
import os
import boto3  # AWS SDK; replace with equivalent for other cloud providers
import anthropic

def get_api_key_from_vault(secret_name: str, region: str = "eu-west-1") -> str:
    """Retrieve the Anthropic API key from AWS Secrets Manager."""
    client = boto3.client("secretsmanager", region_name=region)
    response = client.get_secret_value(SecretId=secret_name)
    # The secret value is a JSON string: {"ANTHROPIC_API_KEY": "sk-ant-..."}
    import json
    return json.loads(response["SecretString"])["ANTHROPIC_API_KEY"]

# Application startup sequence
api_key = get_api_key_from_vault("prod/anthropic-api-key")
llm_client = anthropic.Anthropic(api_key=api_key)
# api_key is now held only in the Anthropic client's internal state;
# do not log it, export it to other modules, or store it in instance variables.
```

Local development environments should use a `.env` file that is explicitly excluded from version control via `.gitignore`. A pre-commit hook that scans staged files for strings matching the `sk-ant-` prefix provides a safety net against accidental commits.

D.9.2 Data Handling, PII, and Financial Regulations

Financial prompts regularly contain information that is legally sensitive: client account numbers (PII), portfolio holdings (potentially subject to fund confidentiality agreements), and in some contexts material non-public information (MNPI). Sending MNPI to an external API raises regulatory concerns under insider trading regulations; even where MNPI is not present, PII transmitted to a third party must be

handled in accordance with GDPR (for EU data subjects) or CCPA (for California residents).

Three mitigations are available, and they are best applied in combination. First, establish a Data Processing Agreement (DPA) with Anthropic under an Enterprise plan that explicitly excludes prompt content from use in model training. This does not resolve MNPI concerns but does address the GDPR processor relationship requirement. Second, apply a named-entity recognition (NER) step to prompts before they are sent, replacing detected PII (names, account numbers, email addresses) with pseudonyms or placeholder tokens:

```
import re

def scrub_pii(text: str) -> str:
    """
    Remove or pseudonymise common PII patterns before sending to the API.
    This is illustrative; a production implementation should use a trained NER
    ↪ model.
    """
    # Replace account numbers (simplified pattern)
    text = re.sub(r'\b[A-Z]{2}\d{10,18}\b', '[ACCOUNT_NUMBER]', text)
    # Replace email addresses
    text = re.sub(
        r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
        '[EMAIL_ADDRESS]',
        text,
    )
    # Replace phone numbers (E.164 format and common variants)
    text = re.sub(r'\+?\d[\d\s\-().]{7,}\d', '[PHONE_NUMBER]', text)
    return text

def build_safe_prompt(raw_prompt: str) -> str:
    return scrub_pii(raw_prompt)
```

Third, for the most sensitive workloads—hedge fund strategies, proprietary risk models, client-specific recommendations—evaluate on-premises model deployment using open-weight models (see Chapter C) as a complement to or replacement for the cloud API. On-premises deployment eliminates data egress entirely at the cost of significantly higher infrastructure and operational overhead.

D.9.3 Audit Trails and Human-in-the-Loop Controls

Regulatory obligations in financial services typically require that every AI-assisted decision or recommendation be traceable after the fact: who requested it, when,

what the model was given, what it produced, and what action was taken. This requires an audit log that is both comprehensive and tamper-resistant.

The audit record introduced in Section D.8.3 provides the raw material. For regulatory purposes, this record should be written to an append-only store: a database table with a write-once policy enforced at the schema level (no `UPDATE` or `DELETE` permissions for the application service account), or an immutable object store such as AWS S3 with Object Lock enabled. The record should include the full prompt (after PII scrubbing), the full response, and the hash of both (for integrity verification).

For high-stakes decisions—trade execution recommendations, credit approvals, portfolio rebalancing proposals—a human-in-the-loop (HITL) gate must be inserted between the model's output and any consequential action. The HITL gate is a review queue: the model's recommendation is written to the queue with a `PENDING` status, and the action is taken only after a human reviewer marks it `APPROVED`. A FastAPI implementation of the HITL pattern:

```python
import uuid, datetime
from enum import Enum
from dataclasses import dataclass, field
from fastapi import FastAPI, HTTPException

app = FastAPI()

class ReviewStatus(str, Enum):
    PENDING   = "PENDING"
    APPROVED  = "APPROVED"
    REJECTED  = "REJECTED"

@dataclass
class PendingRecommendation:
    id:               str
    ticker:           str
    recommendation:   str   # "BUY" | "HOLD" | "SELL"
    rationale:        str
    target_price:     float | None
    created_at:       str
    status:           ReviewStatus = ReviewStatus.PENDING
    reviewed_by:      str | None = None
    reviewed_at:      str | None = None

# In-memory store for illustration; replace with a persistent database.
review_queue: dict[str, PendingRecommendation] = {}

@app.post("/recommendations/submit")
async def submit_recommendation(rec: dict) -> dict:
    """Submit an AI-generated recommendation for human review."""
```

```
    item = PendingRecommendation(
        id=str(uuid.uuid4()),
        ticker=rec["ticker"],
        recommendation=rec["recommendation"],
        rationale=rec["rationale"],
        target_price=rec.get("target_price"),
        created_at=datetime.datetime.utcnow().isoformat(),
    )
    review_queue[item.id] = item
    return {"id": item.id, "status": item.status}

@app.post("/recommendations/{rec_id}/approve")
async def approve_recommendation(rec_id: str, reviewer_id: str) -> dict:
    """Human reviewer approves a pending recommendation."""
    if rec_id not in review_queue:
        raise HTTPException(status_code=404, detail="Recommendation not found.")
    item = review_queue[rec_id]
    if item.status != ReviewStatus.PENDING:
        raise HTTPException(status_code=409, detail="Already reviewed.")
    item.status = ReviewStatus.APPROVED
    item.reviewed_by = reviewer_id
    item.reviewed_at = datetime.datetime.utcnow().isoformat()
    # Downstream: trigger trade execution, notify portfolio manager, etc.
    return {"id": rec_id, "status": item.status}
```

The HITL gate does not eliminate the risk of acting on a flawed recommendation, but it inserts a moment of deliberate human judgement that satisfies supervisory requirements and provides a natural point at which the reviewer can flag anomalies for investigation.

D.10 Applied Example: An Automated Financial Report Generator

This section assembles the primitives from the preceding sections into a complete, end-to-end system. The system accepts a watchlist of stock tickers submitted by a user via a REST endpoint, fans out to a set of analysis subagents, synthesises the results into a formatted document, and returns a finished analyst report—all without human intervention. It is a realistic, production-grade architecture; the components described here are all used in commercial deployments of varying scale.

The system is intentionally modest in scope: it uses three tickers (AAPL, MSFT, NVDA), a simplified data warehouse stub, and a text-only output format. Extending it to support additional tickers, a real warehouse query layer, and PDF rendering via LaTeX or a reporting library is a matter of replacing the stubs with real implementations; the orchestration logic is unchanged.

D.10.1 Architecture Overview

The report generator is structured as a five-stage pipeline. In the first stage, the user submits a watchlist of tickers through a POST endpoint. In the second stage, an orchestrator function fans out to one analysis subagent per ticker, executing all subagents in parallel using `asyncio.gather`. Each subagent retrieves financial data through tool calls, reasons over the data, and returns a `ReportSection` dataclass containing the analyst note, the recommendation, and the price target.

In the third stage, the orchestrator collects all sections and passes them to a formatting agent (a single Claude call with the Sonnet model) that arranges the sections into a coherent document structure with an executive summary and a ranking of the tickers by recommendation strength. In the fourth stage, a validation pass checks each section for internal consistency—data citations present, price targets within plausible ranges—and flags any anomalies for the optional human review queue. In the fifth stage, the validated document is returned to the caller as a structured JSON object that can be rendered as HTML, a PDF, or a spreadsheet by the calling application.

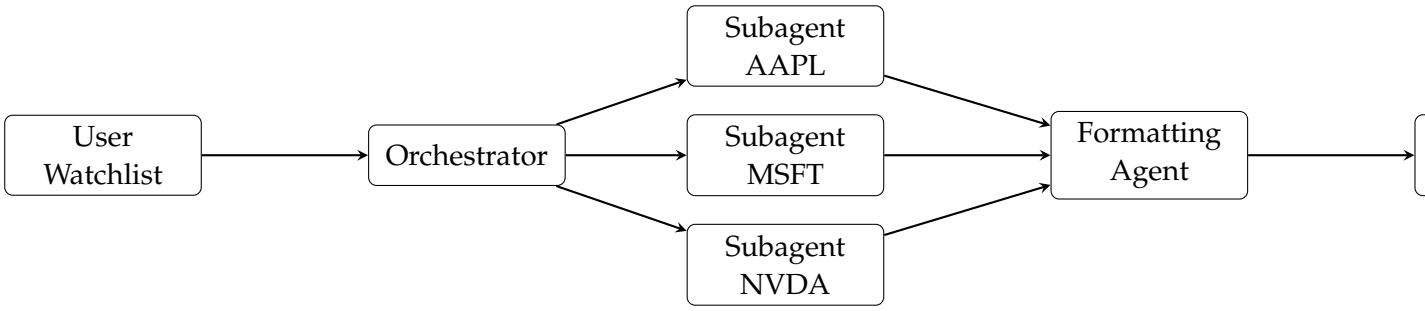

Figure D.1. Architecture of the automated financial report generator. The orchestrator fans out to one subagent per ticker in parallel; the formatting agent and validation pass run sequentially on the aggregated results.

D.10.2 Implementation Walkthrough

The implementation begins with the data schema. Using typed dataclasses enforces discipline about what each component produces and consumes, catches type mismatches at development time, and makes the audit record self-documenting:

```
from dataclasses import dataclass, field

@dataclass
class ReportSection:
    ticker:         str
    company_name:   str
```

```
    analyst_note:   str             # ~300-word prose analysis
    recommendation: str             # "BUY" | "HOLD" | "SELL"
    target_price:   float | None    # 12-month price target in USD
    raw_data:       dict = field(default_factory=dict)   # for audit
    input_tokens:   int = 0
    output_tokens:  int = 0

@dataclass
class FinancialReport:
    sections:            list[ReportSection]
    executive_summary:   str
    ranked_tickers:      list[str]  # best to least attractive
    total_input_tokens:  int = 0
    total_output_tokens: int = 0
    estimated_cost_usd:  float = 0.0
    validation_flags:    list[str] = field(default_factory=list)
```

The analysis subagent is an async function that calls the Opus model with a finance- specific system prompt and a structured user message containing the ticker's financial data. The function parses the recommendation label from the response text using a simple priority search and constructs a `ReportSection`:

```
import asyncio, anthropic, json

client = anthropic.AsyncAnthropic()

async def analyse_ticker(ticker: str, data: dict) -> ReportSection:
    response = await client.messages.create(
        model="claude-opus-4-8",
        max_tokens=1024,
        system=(
            "You are a sell-side equity analyst at a bulge-bracket bank. "
            "Write a concise 300-word analyst note based strictly on the "
            "provided financial data. Structure your note as follows: "
            "one paragraph on revenue and growth trends, one paragraph on "
            "profitability and balance sheet, one paragraph on valuation and risks. "
            "End with a one-sentence investment recommendation containing the words "
            "BUY, HOLD, or SELL in capitals, followed by a 12-month price target "
            "in the format 'Target: $XXX'."
        ),
        messages=[{
            "role": "user",
            "content": (
                f"Ticker: {ticker}\n"
                f"Company: {data.get('name', ticker)}\n"
```

```
                f"Financial data:\n{json.dumps(data, indent=2)}"
            ),
        }],
    )

    text = response.content[0].text

    # Parse recommendation label.
    rec = "HOLD"
    for label in ["BUY", "SELL", "HOLD"]:
        if label in text.upper():
            rec = label
            break

    # Parse target price.
    import re
    target_match = re.search(r"Target:\s*\$(\d+(?:\.\d+)?)", text)
    target = float(target_match.group(1)) if target_match else None

    return ReportSection(
        ticker=ticker,
        company_name=data.get("name", ticker),
        analyst_note=text,
        recommendation=rec,
        target_price=target,
        raw_data=data,
        input_tokens=response.usage.input_tokens,
        output_tokens=response.usage.output_tokens,
    )


async def generate_report(tickers: list[str]) -> FinancialReport:
    # Fetch data for all tickers (stub; replace with warehouse query).
    data_map = {t: fetch_financial_data(t) for t in tickers}

    # Fan out to subagents.
    sections = list(await asyncio.gather(*[
        analyse_ticker(t, data_map[t]) for t in tickers
    ]))

    # Orchestrator synthesis: executive summary and ranking.
    combined = "\n\n".join(
        f"## {s.ticker}: {s.recommendation} | Target ${s.target_price}\n"
        f"{s.analyst_note}"
        for s in sections
    )
    synthesis_response = await client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=512,
        system=(
```

```
            "You are the head of equity research. Given individual analyst notes, "
            "write a two-paragraph executive summary of the overall outlook and "
            "rank the tickers from most to least attractive. "
            "Return a JSON object with keys 'executive_summary' (string) "
            "and 'ranked_tickers' (list of ticker strings)."
        ),
        tools=[{
            "name": "submit_report_metadata",
            "description": "Submit the executive summary and ticker ranking.",
            "input_schema": {
                "type": "object",
                "properties": {
                    "executive_summary": {"type": "string"},
                    "ranked_tickers":    {"type": "array",
                                          "items": {"type": "string"}},
                },
                "required": ["executive_summary", "ranked_tickers"],
            },
        }],
        tool_choice={"type": "tool", "name": "submit_report_metadata"},
        messages=[{"role": "user", "content": combined}],
    )

    meta = synthesis_response.content[0].input

    total_in  = sum(s.input_tokens  for s in sections)
    total_out = sum(s.output_tokens for s in sections)
    total_in  += synthesis_response.usage.input_tokens
    total_out += synthesis_response.usage.output_tokens
    cost = _estimate_cost("claude-opus-4-8", total_in, total_out)

    return FinancialReport(
        sections=sections,
        executive_summary=meta["executive_summary"],
        ranked_tickers=meta["ranked_tickers"],
        total_input_tokens=total_in,
        total_output_tokens=total_out,
        estimated_cost_usd=cost,
    )
```

D.10.3 Evaluation and Quality Gates

Before delivering the report to the user, a validation pass inspects each section for common failure modes. Three checks are applied. The first verifies that each analyst note contains at least one explicit reference to a data point from `raw_data`: a note that does not cite any figures may be hallucinating. The check is performed by a Claude call that is given the note and the raw data and asked to confirm whether the note is grounded.

The second check verifies that each price target is within a plausible range relative to the current market price. A target that is more than 50% above or below the current price is flagged as a potential hallucination and routed to the human review queue rather than included in the delivered report without comment. In practice, genuine analyst targets occasionally exceed this range, but flagging them for review rather than silently including them ensures that anomalous outputs receive scrutiny.

```
def validate_section(section: ReportSection) -> list[str]:
    """
    Return a list of validation flags for the section.
    An empty list means the section passed all checks.
    """
    flags = []

    # Check 1: at least one specific data point cited.
    data_values = [str(v) for v in section.raw_data.values()
                   if isinstance(v, (int, float, str))]
    if not any(val[:6] in section.analyst_note for val in data_values):
        flags.append(
            f"{section.ticker}: analyst note may not cite specific data from
            ↪ warehouse."
        )

    # Check 2: price target plausibility.
    if section.target_price is not None:
        current_price = section.raw_data.get("current_price")
        if current_price and isinstance(current_price, (int, float)):
            ratio = section.target_price / current_price
            if ratio > 1.5 or ratio < 0.5:
                flags.append(
                    f"{section.ticker}: target price ${section.target_price:.2f}
                    ↪ "
                    f"is outside ±50%% of current price ${current_price:.2f}. "
                    "Flagged for human review."
                )

    # Check 3: recommendation label present.
    if section.recommendation not in ("BUY", "HOLD", "SELL"):
        flags.append(
            f"{section.ticker}: recommendation label '{section.recommendation}' "
            "not recognised."
        )

    return flags

def validate_report(report: FinancialReport) -> FinancialReport:
    """Run validation on all sections and attach flags to the report."""
    all_flags = []
```

```
    for section in report.sections:
        all_flags.extend(validate_section(section))
    report.validation_flags = all_flags
    return report
```

The third quality gate is a cost audit. After the report is generated, the orchestrator logs the total token spend and estimated cost per report run. If the cost exceeds a configurable threshold (for example, $0.50 per report), an alert is raised so that the engineering team can investigate whether a prompt has grown unexpectedly large or whether a model has entered a high-token-consumption loop. Over time, cost-per-report trends reveal whether system changes are moving costs in the intended direction and whether the model-tier selection remains optimal as usage patterns evolve.

The complete system, assembled from the components above, processes a three-ticker watchlist in approximately twelve to fifteen seconds of wall-clock time (dominated by the parallel subagent calls, each of which takes four to six seconds for Opus at current generation speeds). The estimated cost per run at three tickers is approximately $0.05 to $0.10, depending on the length of the financial data provided to each subagent. Scaling to twenty tickers increases wall-clock time by less than 50% (because the subagents run in parallel) and cost linearly.

APPENDIX E

GIT AND GITHUB: VERSION CONTROL FOR AI-ASSISTED PROJECTS

Remark E.1 (Learning Objectives). After working through this appendix, the reader should be able to:

1. Explain why version control is especially important when working with AI coding assistants, and describe the specific failure modes it mitigates.
2. Initialise a Git repository, stage and commit changes, and interpret the output of `git status`, `git log`, and `git diff`.
3. Use branching and merging to isolate AI-generated work from stable code, and recover from unwanted AI changes using `git checkout` and `git revert`.
4. Push a repository to GitHub, open a pull request, and link AI commit history to project milestones.
5. Apply a disciplined commit-per-step workflow that makes AI contributions reviewable, attributable, and reversible.

E.1 Why Version Control Matters for AI-Assisted Work

THE BIGGER PICTURE

AI coding assistants—Claude Code, Cline, GitHub Copilot, and their successors—share a property that distinguishes them from human collaborators: they can generate large volumes of code changes in seconds, across many files simultaneously, with plausible-looking output that may nonetheless contain subtle errors. A traditional code review catches one human's typo or logic error across a handful of files. An AI session might touch fifty files in three minutes. Without version control, reviewing, attributing, and reversing those changes is nearly impossible.

Version control systems record the *history* of every change to every file in a

project. Git, the dominant distributed version control system, does this by storing a directed acyclic graph of *commits*—snapshots of the project at a point in time—linked by parent pointers. Each commit carries a message, an author, a timestamp, and a cryptographic hash of the entire project state. The result is an audit log: one can determine exactly what changed, when, and in which session—including AI-assisted sessions.

This audit log serves four functions when working with AI:

1. **Reversibility.** If an AI agent introduces a regression—a previously passing test now fails, or a file is unintentionally overwritten—`git revert` or `git reset` can restore any prior state in seconds. Without Git, recovery requires manual reconstruction.
2. **Attribution.** Knowing that a specific function was written by an AI in session 7 of a project, rather than by a human collaborator, carries interpretive weight during code review. AI-generated code deserves additional scrutiny for hallucinated API calls, incorrect assumptions about library versions, and subtle logic errors that read fluently but execute incorrectly.
3. **Incremental review.** A commit containing ten lines of change is reviewable in minutes. A commit containing two thousand lines across thirty files is not. Disciplined small commits—one per completed sub-task—keep AI contributions within human review bandwidth.
4. **Collaboration.** Multiple researchers working in parallel on different parts of a project, some using AI assistants and some not, can merge their work without conflict using Git's branch-and-merge model.

Remark E.2 (Git is not a backup system). Git tracks changes to text files efficiently but is not designed for large binary files (datasets, compiled models, PDF outputs). For research projects, keep `*.csv`, `*.parquet`, `*.pkl`, and `*.h5` files out of the repository by listing them in `.gitignore`, and store them separately in a data store (institutional storage, S3, DVC). This keeps repositories fast and readable.

E.2 Installation and First Repository

E.2.1 Installing Git

Git is pre-installed on macOS (via Xcode Command Line Tools) and most Linux distributions. On Windows, download Git for Windows from https://git-scm.com

or install via `winget`:

```
winget install --id Git.Git -e --source winget
```

Verify the installation:

```
git --version
```

Configure your identity—every commit is tagged with these values:

```
git config --global user.name  "Jane Smith"
git config --global user.email "jane.smith@finance-lab.edu"
```

E.2.2 Initialising a Repository

To place an existing project directory under version control:

```
cd my-finance-project
git init
```

This creates a hidden `.git/` directory that stores the entire history. The working directory is unchanged.

For a new project, it is more common to create the repository on GitHub first and then clone it locally (Section E.6).

E.3 The Core Workflow: Stage, Commit, Push

THE BIGGER PICTURE

The fundamental Git loop is:

1. **Work.** Edit files in the working directory—or let an AI assistant edit them.
2. **Stage.** Select the changes you want to record with `git add`.

3. **Commit.** Snapshot the staged changes with a descriptive message using `git commit`.
4. **Push.** Upload commits to a remote (typically GitHub) with `git push`.

This cycle should be tight when working with AI. The recommended discipline is: *commit after every completed sub-task*, not after every session. If an AI agent refactors a function, add and commit before asking it to do the next task. This makes each commit small, reviewable, and reversible.

E.3.1 Checking Status and Inspecting Changes

```
git status           # which files have changed since last commit?
git diff             # show unstaged line-level changes
git diff --staged    # show staged line-level changes
```

The output of `git diff` is a *unified diff*: lines prefixed with - were removed, lines prefixed with + were added. This is the primary interface for reviewing AI-generated changes before committing them.

E.3.2 Staging and Committing

Stage specific files (preferred) or all changed files:

```
git add src/sentiment.py tests/test_sentiment.py   # specific files
git add .                                           # all changes (use with care)
```

Commit with a message that explains *why* the change was made, not just what:

```
git commit -m "feat: replace regex tokeniser with spaCy for sentence splitting"
```

A widely adopted convention for commit message prefixes is:

Prefix	Meaning
`feat:`	New feature or capability
`fix:`	Bug fix
`refine:`	Improvement without new functionality
`chore:`	Maintenance (dependencies, config)
`docs:`	Documentation only
`test:`	Tests only

When using AI assistants, add a `Co-Authored-By` trailer to the commit message to attribute the AI contribution:

```
git commit -m "feat: add FinBERT sentiment pipeline

Co-Authored-By: Claude Sonnet 4.6 <noreply@anthropic.com>"
```

E.3.3 Viewing History

```
git log --oneline          # compact one-line summary per commit
git log --stat             # show files changed per commit
git log -p                 # show full diff for each commit
git log --author="Claude"  # filter by author (or Co-Authored-By)
```

Table E.1 shows a representative `git log -oneline` output from a finance research project that used an AI assistant.

Table E.1. Example `git log` output from an AI-assisted finance project. Each short hash identifies a unique commit; messages distinguish human-authored from AI-assisted work.

Hash	Message
`a3f7c2e`	feat: add FactSet API integration (human)
`9b1d04a`	feat: extend sentiment model to earnings calls (AI-assisted)
`3e8c17f`	fix: correct EDGAR filing date parsing for 10-Qs (AI-assisted)
`c2a5b89`	chore: update requirements.txt
`7f3d12e`	feat: initial FinBERT pipeline scaffold (AI-assisted)
`b0f9e3c`	chore: git init, add .gitignore

E.4 Undoing Changes

THE BIGGER PICTURE

Undoing changes is one of the most important capabilities for AI-assisted development. AI agents occasionally produce code that passes a superficial review but fails in production, or silently introduces incorrect logic. The ability to return to a known good state—without manual reconstruction—is what makes it safe to let an AI agent work autonomously for an extended period.

E.4.1 Before Staging: Discarding Working-Directory Changes

```
git restore path/to/file.py          # discard one file's changes
git restore .                        # discard all unstaged changes
```

These commands are *destructive*: unsaved working-directory changes are gone permanently. Use them when an AI has produced a bad edit and you want to start over.

E.4.2 After Committing: Reverting and Resetting

```
# Safe: create a new commit that undoes the changes of commit <hash>
git revert <hash>

# More powerful: move HEAD back to <hash>, discarding later commits
# --soft keeps changes staged; --mixed keeps them unstaged; --hard discards them
git reset --hard <hash>
```

Prefer `git revert` for work that has been pushed to a shared remote, because it adds to history without rewriting it. Use `git reset` only on local-only commits.

Remark E.3 (Checkpoint discipline with Claude Code). Claude Code takes a Git snapshot before and after every tool call that writes a file. These checkpoints are stored in a shadow repository inside the session directory and can be browsed with `/diff` and restored with `/revert` within a session. However, they are *not* visible to ordinary `git log` commands on the project repository. The recommended practice is to commit to the project repository at meaningful boundaries—not to rely on Claude Code's internal checkpoints as the sole audit trail.

E.5 Branches

A *branch* is a named pointer to a commit. Creating a branch is instantaneous and costs nothing; it is the standard mechanism for isolating experimental or AI-generated work from stable code.

```
git branch ai/edgar-sentiment    # create a branch
git switch ai/edgar-sentiment    # move to it (also: git checkout)
# ... let the AI work here ...
git switch main                  # return to main
git merge ai/edgar-sentiment     # merge if the work is good
git branch -d ai/edgar-sentiment   # delete the branch
```

The naming convention `ai/<task>` clearly marks branches that contain primarily AI-generated work and flags them for extra scrutiny during review.

If a merge produces *conflicts*—both the branch and main modified the same lines—Git marks the file with conflict markers:

```
<<<<<<< HEAD
return df.groupby("ticker").mean()          # your version
=======
return df.groupby("ticker").mean(numeric_only=True)  # AI version
>>>>>>> ai/edgar-sentiment
```

Edit the file to the correct result, remove the markers, stage the file, and complete the merge with `git commit`.

E.6 GitHub: Remote Repositories and Collaboration

GitHub (https://github.com) hosts Git repositories in the cloud and adds collaboration features: pull requests, code review, issue tracking, and CI/CD integration.

E.6.1 Creating a Repository on GitHub

1. Log into GitHub and click **New repository**.
2. Choose a name, select **Private** for unpublished research, and optionally add a `README.md` and `.gitignore`.
3. Copy the HTTPS or SSH remote URL.

Link the local repository to the remote and push:

```
git remote add origin https://github.com/username/finance-project.git
git push -u origin main
```

Subsequently, `git push` and `git pull` require no further arguments (the tracking branch is set by `-u`).

E.6.2 Pull Requests

A *pull request* (PR) is a GitHub interface for proposing that a branch be merged into another. For AI-assisted work, the PR is the natural review checkpoint:

1. Push the AI-work branch to GitHub: `git push -u origin ai/edgar-sentiment`.
2. Open a PR on GitHub: **Compare & pull request** → write a description of what the AI produced and what you verified.
3. Review the diff: every file touched by the AI is visible side-by-side with the previous version. Add line-level comments where behaviour is unclear.
4. Merge when satisfied; delete the branch.

The PR thread becomes a permanent record of what the AI produced, what questions were raised during review, and what changes were requested before merge.

E.6.3 The .gitignore File

The `.gitignore` file tells Git to ignore matching paths. A sensible starting template for a Python finance project:

```
# Python artefacts
__pycache__/
*.pyc
*.pyo
.venv/
*.egg-info/

# Jupyter
.ipynb_checkpoints/

# Data (store externally)
data/raw/
```

```
data/processed/
*.csv
*.parquet
*.h5

# API keys and secrets
.env
secrets.toml
*.key

# IDE
.vscode/settings.json
.idea/

# OS
.DS_Store
Thumbs.db
```

Never commit files containing API keys, passwords, or proprietary data. Use environment variables or a secrets manager instead.

E.7 A Disciplined Workflow for AI-Assisted Finance Research

THE BIGGER PICTURE

The following workflow synthesises the preceding sections into a repeatable practice for using AI coding assistants on research projects.

1. **Start on a branch.** Before starting an AI session, create a branch: `git switch -c ai/`*`task-name`*. The branch name documents the purpose of the session.
2. **State the task clearly.** Write the task description in the AI prompt and also in a commit message or issue. A clear record of what the AI was asked to do is as important as a record of what it did.
3. **Commit after each sub-task.** When the AI completes a self-contained piece of work—a function, a test, a data pipeline step—stop, review the diff (`git diff`), and commit. Do not let uncommitted AI changes accumulate across long sessions.
4. **Review before merging.** Open a PR even when working alone. The GitHub diff view is the most effective format for reviewing AI-generated code because it enforces line-by-line attention.

5. **Tag stable milestones.** Use `git tag v1.0` to mark commits that correspond to submitted paper drafts, reproducibility checkpoints, or data collection cut-offs. Tags survive branch deletion and provide unambiguous version references for citations and replication.
6. **Document the AI tools used.** Add a short `AI.md` file to the repository noting which AI tools were used, for which tasks, and during which date ranges. As AI tool capabilities evolve rapidly, future readers will benefit from knowing which version of which tool produced which parts of the codebase.

APPENDIX F

CLINE: A PRACTITIONER'S REFERENCE

Remark F.1 (Learning Objectives). After working through this appendix, the reader should be able to:

1. Install Cline as a VS Code extension, configure an LLM provider API key, and start a first task session.
2. Explain Cline's Plan/Act execution model and use Plan Mode to review a proposed plan before any files are modified.
3. Configure a `.clinerules` file to encode project-specific conventions into every Cline session.
4. Use Cline's checkpoint system to inspect and roll back AI-generated changes to any prior state within a session.
5. Select an appropriate LLM provider and model tier for a given finance research task, balancing accuracy, context length, and API cost.
6. Identify at least three quantitative-finance research tasks that map naturally to Cline workflows, and describe how to structure the corresponding prompts and permission settings.

F.1 What is Cline

THE BIGGER PICTURE

Cline is an open-source autonomous AI coding agent delivered as a Visual Studio Code extension. It was created by developer Saoud Rizwan and first released in June 2024 under the name *Claude Dev* as part of Anthropic's "Build with Claude" contest. In October 2024 the project was renamed *Cline* (version 2.0), incorporated as Cline Bot Inc., and opened to the broader AI community. By mid-2026, the extension had accumulated over 8 million installs across VS Code and JetBrains, with contributions from more than 250 open-source contributors. The full source code is available at `https://github.com/cline/`

`cline` under the Apache 2.0 licence.

The defining characteristic of Cline is its *model-agnostic* architecture: unlike Claude Code, which is tied to Anthropic models, Cline treats the LLM as a replaceable component and supports more than thirty provider backends, including Anthropic, OpenAI, Google, AWS Bedrock, Azure OpenAI, DeepSeek, and local models via Ollama. The developer supplies their own API key; there is no bundled model subscription.

For finance researchers, this design has two practical implications. First, it makes Cline suitable for environments where model-provider choice is constrained by institutional compliance policy. Second, it exposes the per-session token cost directly: every API call is visible, attributable, and billable, which encourages token-efficient prompting.

F.1.1 Positioning Among AI Coding Assistants

Table F.1 compares Cline and Claude Code across the dimensions most relevant to academic finance research.

Table F.1. Cline versus Claude Code: key dimensions for finance research use.

Dimension	Cline	Claude Code
Interface	VS Code extension; JetBrains; CLI	Terminal (CLI), IDE panels
LLM support	30+ providers (model-agnostic)	Anthropic models only
Licence	Apache 2.0 open source	Proprietary (Anthropic)
Pricing	Free extension; BYOK API usage	$20/mo Max or API usage
Browser automation	Yes (Puppeteer/Computer Use)	No
MCP support	Yes (first-class)	Yes
Checkpoints	Shadow Git repo per tool call	Session-scoped snapshots
Codebase indexing	None (dynamic file reads)	Whole-project read on start
`.rules` file	`.clinerules` per workspace	`CLAUDE.md` per project

Many practitioners use both tools: Cline for IDE-embedded development with visual diff review and browser-verified UI work; Claude Code for complex terminal-native tasks that benefit from its whole-project context and richer skill system.

F.2 Installation and Setup

F.2.1 Prerequisites

Cline requires:

- Visual Studio Code 1.84 or later (download from https://code.visualstudio.com).
- An API key from at least one supported LLM provider.
- (Optional) Node.js 18+ if you plan to use the Cline CLI separately.

F.2.2 Installing the Extension

Install Cline from the VS Code Marketplace:

1. Open VS Code and press `Ctrl+Shift+X` (Windows/Linux) or `Cmd+Shift+X` (macOS) to open the Extensions panel.
2. Search for **Cline** (publisher ID: `saoudrizwan.claude-dev`).
3. Click **Install**.

Alternatively, install from the command line:

```
code --install-extension saoudrizwan.claude-dev
```

After installation, the Cline icon appears in the VS Code activity bar. Click it to open the Cline panel.

F.2.3 Configuring a Provider

On first launch, Cline prompts for an API provider and key. Click the settings gear in the Cline panel to open the provider configuration:

1. Select a provider from the dropdown (e.g. **Anthropic**).
2. Enter your API key (e.g. `sk-ant-...`).
3. Select the model (e.g. `claude-sonnet-4-6`).
4. Click **Save**.

API keys are stored in VS Code's secret storage (OS keychain), not in plain text on disk. For team environments, use a `.env` file and configure Cline to read it from the project root.

Table F.2 summarises the most commonly used providers for finance research workloads.

Table F.2. Selected LLM providers available in Cline, with notes for finance research use.

Provider	Models	Notes
Anthropic	Claude Sonnet, Opus, Haiku	Strongest reasoning; 200K token context; BYOK
OpenAI	GPT-4o, o1, o3-mini	Broad library support; function calling
Google	Gemini 1.5 Pro/Flash	1M token context; useful for very long filings
DeepSeek	DeepSeek-V3, R1	Low cost; strong at code generation
Ollama	Any GGUF model	Fully local; no data leaves the machine
AWS Bedrock	Claude, Llama, Titan	VPC-isolated; suitable for regulated data

F.3 The Plan/Act Execution Model

THE BIGGER PICTURE

Cline's most important safety feature is its *Plan/Act* distinction. Before making any change to the file system or running any command, Cline can operate in Plan Mode, where it reads files and reasons about what it would do—without doing it. The developer reviews and approves the plan, then switches to Act Mode to execute. This separation is especially valuable for finance research, where a misunderstood task can corrupt a dataset or delete months of carefully assembled code.

F.3.1 Plan Mode

Enable Plan Mode by toggling the **Plan** button in the Cline panel before submitting a task. In Plan Mode, Cline:

- Reads all files it determines are relevant to the task.
- Constructs a step-by-step plan describing every file it intends to create, modify, or delete and every command it intends to run.

- Presents the plan as a structured document in the chat panel.
- Takes no action on the file system.

Review the plan carefully. Common issues to check:

- Does the plan modify files outside the expected scope?
- Does it propose to run commands that could have side effects (deleting data, pushing to remote, installing packages globally)?
- Are the proposed file paths consistent with the project structure?

Once the plan is satisfactory, click **Approve** (or toggle to Act Mode and submit). Cline executes the plan step by step, pausing for approval at each tool call if the **Auto-approve** setting is off.

F.3.2 Act Mode and Tool Calls

In Act Mode, Cline issues *tool calls* to interact with the environment. Each tool call is displayed in the Cline panel before execution, giving the developer an opportunity to approve or reject it. The built-in tools are:

`read_file` Read the content of a file. Cline uses this extensively to understand the codebase before making changes.

`write_file` Write a new file or overwrite an existing one. Cline shows the proposed content in a diff view; the developer approves or edits before the file is written.

`replace_in_file` Make targeted edits to an existing file using `<SEARCH>`/`<REPLACE>` blocks, analogous to Claude Code's Edit tool.

`execute_command` Run a shell command. Output is streamed in real time. Cline iterates on failures by reading error output and attempting corrections.

`browser_action` Launch a headless Chromium browser, navigate to a URL, click elements, take screenshots, and read the rendered page. This enables automated testing of web-based dashboards and data portals.

`use_mcp_tool` Call a tool exposed by any connected MCP server, extending Cline's reach to external databases, APIs, and custom tools.

F.3.3 Auto-Approve Settings

In the Cline settings panel, individual tool categories can be set to auto-approve, reducing friction for trusted operations:

- **Read files**: safe to auto-approve for most projects.
- **Write files**: approve manually during early exploration; auto-approve after establishing trust in the agent's file-scope discipline.
- **Execute commands**: approve manually unless the command set is tightly constrained by a `.clinerules` allowlist.
- **Browser actions**: approve manually; browser automation can interact with authenticated sessions and is high-consequence.

F.4 Project Configuration with `.clinerules`

The `.clinerules` file, placed at the project root, is injected into Cline's system prompt at the start of every session. It is the Cline equivalent of Claude Code's `CLAUDE.md`. Use it to encode:

- Project structure and conventions (file naming, directory layout).
- Coding standards (PEP 8 for Python, type hints, docstring format).
- Constraints on AI behaviour ("never modify files in `data/raw/`", "always run `pytest` before committing").
- Domain context ("this project uses BibTeX for references", "financial figures are always in USD millions unless noted").

```
# Finance Research Lab - Cline Rules

## Project Structure
- `src/` - production Python modules
- `notebooks/` - Jupyter exploratory notebooks (never import from here)
- `data/raw/` - read-only; never modify or delete
- `data/processed/` - output of pipeline scripts
- `tests/` - pytest suite; maintain >80% coverage

## Coding Standards
- Python 3.11+; use type hints on all public functions
- Format with `black`; lint with `ruff`
- Run `pytest` before any commit

## Constraints
- Do not install packages not already in `requirements.txt`
  without explicit approval
- Do not push to remote without explicit instruction
- Do not write API keys or credentials into any file

```

```
## Domain Context
- All monetary amounts in USD millions unless stated otherwise
- EDGAR filings are retrieved via the SEC REST API; respect 10 req/s rate limit
- Citation format is BibTeX authoryear (biblatex)
```

F.5 Checkpoints and Recovery

Every time Cline writes or modifies a file, it records a *checkpoint* in an internal shadow Git repository maintained inside the VS Code extension's data directory. Checkpoints are independent of the project's own Git repository.

To use checkpoints:

1. In the Cline chat history, each tool call that modified a file shows a **View diff** link. Click it to see exactly what the tool call changed.
2. To revert the project to the state before a specific tool call, click the **Restore checkpoint** button next to that tool call.
3. All file writes made after that tool call are undone; the project returns to exactly the checkpoint state.

Remark F.2 (Checkpoints vs. project Git history). Cline's checkpoints are granular (one per tool call) and scoped to the current VS Code window. They are a session-level safety net, not a long-term audit trail. After a session, commit the desired changes to the project repository with a descriptive message that attributes the AI contribution. The project Git history is the durable record; checkpoints are disposable scaffolding.

F.6 Context Management

Cline does not index the codebase upfront or build a vector store. Instead, it reads files dynamically as needed—an explicit architectural choice modelled on how a senior engineer would approach an unfamiliar repository. When the context window fills, Cline truncates the oldest messages. The developer can also trigger manual context management:

`/newtask` Summarises the current conversation into a structured handoff note (plan, work completed, files changed, next steps) and starts a fresh session with a clean context window. Use this when a long session is approaching the context limit or when switching to a new sub-task.

`/smol` Compresses the chat history by summarising older turns without starting a new task.

Remark F.3 (Token costs for finance workloads). A typical Cline session on a finance data pipeline project—reading several Python modules, running tests, and iterating on a bug fix—uses between 50,000 and 200,000 tokens. At Claude Sonnet rates (approximately \$3/M input, \$15/M output as of mid-2026), this corresponds to roughly \$0.50–\$2.00 per session. Heavy users find flat-rate subscriptions more economical; light academic users benefit from the pay-as-you-go model.

F.7 Slash Commands and Context Injection

Table F.3. Commonly used Cline slash commands and context operators.

Command / Operator	Effect
`/newtask`	Summarise session and start fresh (preserves handoff note)
`/smol`	Compress context in place
`/newrule`	Open the `.clinerules` file for editing
`/reportbug`	Open a GitHub issue on the Cline repository
`@`*`file`*	Include a specific file in the next message context
`@`*`folder`*	Include all files in a folder
`@url`	Fetch and include the content of a URL
`@problems`	Include the current VS Code diagnostics (linter errors)

The `@problems` operator is particularly useful in finance data pipelines: pasting the current set of type errors or linter warnings as context often allows Cline to fix them in a single tool call without reading the entire file.

F.8 MCP Integration

Cline supports the Model Context Protocol (MCP), a standard interface for connecting AI agents to external tools, databases, and APIs. MCP servers can be installed from the Cline MCP marketplace (accessible from the settings panel) or configured manually in `cline_mcp_settings.json`.

Finance-relevant MCP servers include:

EDGAR MCP Queries the SEC EDGAR REST API and returns structured filing data—accession numbers, document links, XBRL financial statements— directly into the Cline context.

SQL MCP Connects to a local or remote database (PostgreSQL, DuckDB, SQLite) and executes queries on behalf of the AI agent, enabling Cline to explore and transform financial datasets without writing data to intermediate files.

Bloomberg Terminal MCP (enterprise) Routes Bloomberg BQL queries through the MCP interface, providing real-time market data to the agent within a session.

F.9 Finance Research Workflows

THE BIGGER PICTURE

The following examples illustrate how Cline's features combine into coherent research workflows. Each workflow begins with a task description submitted in the Cline chat panel.

F.9.1 EDGAR Data Pipeline

Task prompt:

```
Using the SEC EDGAR REST API, build a Python function that:
1. Accepts a list of CIK numbers and a date range
2. Retrieves all 10-K filings for each company in that range
3. Extracts the MD&A section (Item 7) from the HTML filing
4. Returns a pandas DataFrame with columns: cik, filing_date, mda_text
Respect the 10 req/s rate limit. Write unit tests.
```

Expected Cline actions: (in Plan Mode)

1. Read `src/edgar.py` (if it exists) to understand existing helpers.
2. Read `requirements.txt` to check available libraries.
3. Propose creating `src/edgar_mda.py` with the function and `tests/test_edgar_mda.py` with unit tests.
4. No files modified until plan approved.

F.9.2 LLM Sentiment Evaluation

Task prompt:

```
Evaluate three sentiment models on the Financial PhraseBank dataset:
FinBERT (local via transformers), GPT-4o-mini (API), and a zero-shot
Claude Haiku prompt. Report accuracy, macro-F1, and cost per 1,000 sentences.
```

Cline advantage over a static notebook: Cline can interleave code execution with plan refinement. If the FinBERT model raises an out-of-memory error on the test machine, Cline reads the error output and proposes batching or an alternative model—within the same session, without manual intervention.

F.9.3 Automated Replication Check

Task prompt:

```
Replicate Table 3 of Loughran & McDonald (2011) using the EDGAR full-text
search index. Use the LM sentiment dictionary from their website. Compare
your coefficients to the published values and flag any discrepancy > 10%.
```

This workflow benefits from Cline's browser automation: it can navigate the LM dictionary download page, trigger the download, verify the file integrity, and proceed with the analysis—all in a single session.

F.10 Security Considerations

1. **Never store API keys in `.clinerules`.** That file is committed to the repository. Use environment variables or VS Code's secret storage.
2. **Review every `execute_command` call before approval.** Commands can exfiltrate data, install packages, or push code to remotes. Enable manual approval for all shell commands in Cline settings.
3. **Use Ollama for sensitive data.** When working with non-public client data or proprietary trading signals, configure Cline to use a locally hosted model via Ollama. No tokens leave the machine.
4. **Audit the MCP server list.** Each MCP server has access to the tools it exposes. Only install MCP servers from trusted sources and review their source code before connecting them to sessions that handle sensitive financial data.

REFERENCES

Anthropic (2024). *Model Context Protocol*. Open standard specification, retrieved May 2026. URL: https://modelcontextprotocol.io.

Anthropic (2025a). *Anthropic API Reference*. Official reference documentation, retrieved June 2026. URL: https://docs.anthropic.com/en/api/.

Anthropic (2025b). *Claude Code SDK*. Official SDK documentation, retrieved June 2026. URL: https://docs.anthropic.com/en/docs/claude-code/sdk.

Anthropic (2025c). *Claude Code: Agentic Coding in Your Terminal*. Official documentation, retrieved May 2026. URL: https://code.claude.com/docs.

Araci, D. (2019). "FinBERT: Financial Sentiment Analysis with Pre-trained Language Models". In: *arXiv preprint arXiv:1908.10063*.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. (2021). "Evaluating Large Language Models Trained on Code". In: *arXiv preprint* arXiv:2107.03374.

Dettmers, T., A. Pagnoni, A. Holtzman, and L. Zettlemoyer (2023). "QLoRA: Efficient Finetuning of Quantized LLMs". In: *arXiv preprint arXiv:2305.14314*.

Gerganov, G. et al. (2023). *llama.cpp: LLM Inference in C/C++*. Retrieved May 2026. URL: https://github.com/ggml-org/llama.cpp.

Hu, E. J., Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen (2022). "LoRA: Low-Rank Adaptation of Large Language Models". In: *International Conference on Learning Representations*.

Jimenez, C. E., J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan (2024). "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *arXiv preprint* arXiv:2310.06770.

Kwon, W., Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica (2023). "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Liu, X. et al. (2024). "AgentBench: Evaluating LLMs as Agents". In: *International Conference on Learning Representations*.

OpenAI (2025). *Introducing Codex*. Retrieved May 2026. URL: https://openai.com/index/introducing-codex/.

OpenAI (2026). *Introducing GPT-5.5*. Retrieved May 2026. URL: https://openai.com/index/introducing-gpt-5-5/.

Park, J. S., J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein (2023). "Generative Agents: Interactive Simulacra of Human Behavior". In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*.

Patil, S. G., T. Zhang, X. Wang, and J. E. Gonzalez (2023). "Gorilla: Large Language Model Connected with Massive APIs". In: *arXiv preprint arXiv:2305.15334*.

Peng, S., E. Kalliamvakou, P. Croft, and M. Descôtes (2023). "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot". In: *arXiv preprint* arXiv:2302.06590.

Schick, T., J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom (2023). "Toolformer: Language Models Can Teach Themselves to Use Tools". In: *Advances in Neural Information Processing Systems* 36.

The Linux Kernel Organization (2022). *Landlock: Unprivileged Access Control for the Linux Kernel*. Linux kernel documentation, retrieved May 2026. URL: https://docs.kernel.org/userspace-api/landlock.html.

Wu, S., O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann (2023). "BloombergGPT: A Large Language Model for Finance". In: *arXiv preprint arXiv:2303.17564*.

Yao, S., J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao (2023). "ReAct: Synergizing Reasoning and Acting in Language Models". In: *International Conference on Learning Representations*.