ChatGPT

AI-Native IDE Implementation Specification (v0.1)

1 Executive Summary

Problem & Opportunity

Modern software development is constrained by tooling that treats AI as an optional add-on rather than a first-class citizen. Developers juggle multiple separate assistants for code completion, chat, edits, refactoring and debugging, and switching contexts is cumbersome. Many assistants are limited by short context windows, lack of privacy controls, or inability to operate across entire repositories and run code safely. A unified AI-native integrated development environment (IDE) can dramatically accelerate development by blending language models, static analysis and agentic workflows into a coherent user experience.

Vision

We propose building **AIDE** (AI-native Development Environment), a full-fledged IDE that matches and surpasses feature parity with Cursor, Windsurf (Codeium), Bolt (StackBlitz), GitHub Copilot, Continue, Cody, Tabnine, JetBrains AI, Amazon Q Developer, Replit AI, and other leading tools. AIDE will seamlessly integrate multiple coding LLMs with automatic routing and user overrides, operate in privacy-preserving modes, support agentic workflows across entire repositories, and allow extensibility via a plugin marketplace. Our goal is a 10× productivity improvement over manual coding by unifying completion, editing, refactoring, testing, debugging and deployment into a single interactive environment.

User Personas

- **Solo Developer** – uses AIDE in local mode to build side projects and values privacy. Needs quick completions, refactoring and tests.
- **Enterprise Engineer** – works on massive monorepos behind corporate firewalls. Requires strict data governance, multi-file editing, agentic automation and audit logs.
- **DevOps / SRE** – uses integrated terminal and scripting to diagnose issues and deploy fixes. Needs safe command planning and rollback.
- **Researcher / Educator** – leverages large-context models for code understanding, debugging and experiment automation.

Top-Level Architecture

The high-level system consists of four planes (see Figure 1):

1. **UI/Editor** – VS Code and JetBrains plugins plus a standalone Tauri/Electron shell provide a responsive, keyboard-centric interface. Components include an inline completion engine, multi-file diff viewer, terminal emulator, chat panel and agent control centre.

2. **AI Services** – a **Model Router & Policy Engine** automatically selects among multiple LLMs (Codestral 25.01, Code Llama 70B, DeepSeek Coder V3, Qwen 2.5 Coder 7B/32B, Llama 3.1 8B/70B/405B and future models) based on task type, context, latency budgets and user preferences. LLM providers may run locally via Ollama or remote via API.
3. **Context & Knowledge Layer** – a **Context Engine** builds embeddings, symbol graphs and code graphs using tree-sitter-based parsers. It stores data in a pluggable vector store (FAISS, PGVector, Milvus, etc.), and performs retrieval-augmented generation (RAG) with AST-based chunking to maintain structural integrity [1].
4. **Execution & Agent Layer** – execution sandboxes (local venvs, Docker, remote VM) run code safely. An agent framework orchestrates tools (file I/O, code search, test runner, command runner) within safe loops. Telemetry, caching and audit pipelines monitor and record interactions.

```
graph TD
    subgraph UI
        Editor[Editor & Chat]
        Terminal[Integrated Terminal]
        AgentPanel[Agent Control]
    end
    subgraph AI
        Router[Model Router & Policy Engine]
        Models[L1..LN LLMs]
        ContextEngine[Context Engine & Vector DB]
        PromptPacker[Prompt Templates & Context Packer]
    end
    subgraph Execution
        Sandbox[Execution Sandboxes (Local/Remote)]
        Agents[Agent Runtime]
        Tools[Plugins & Tools]
    end
    subgraph Infra
        Indexer[File Indexer & Watchman]
        Telemetry[Telemetry & Observability]
        Secrets[Secrets & Policy Store]
    end
    Editor <--> Router
    Router <--> Models
    Router --> PromptPacker
    PromptPacker --> Models
    Editor --> ContextEngine
    ContextEngine --> PromptPacker
    Agents <--> Sandbox
    Agents <--> Tools
    Editor <--> Agents
    Indexer --> ContextEngine
    Telemetry --> Router
    Telemetry --> Agents
    Secrets --> Router
    Secrets --> Agents
```

Figure 1 – High-level architecture.

Differentiators vs. Existing Tools

Differentiator	Description
Unified Agentic IDE	AIDE unifies completion, chat, editing, debugging, testing and deployment into a cohesive UI, avoiding context-switching between separate tools.
Model Routing & Ensembling	A policy engine chooses among multiple LLMs based on task, language and cost. Users can override globally or per task.
Repo-Scale RAG	Index entire repositories, build AST-aware chunks and call graph cross-references. Retrieval includes embeddings, lexical search and code graph features [1] .
Autonomous Workflow Agents	Planner → Coder → Executor → Tester → Critic → Reviewer agents automate multi-step tasks, run tests and create pull requests while keeping developers in the loop.
Observable Context & Safety	Users can inspect the exact context packets sent to models, with redaction of secrets; prompts and responses are logged for audit.
Privacy Modes	Local (air-gapped), Hybrid and Cloud modes control where code and telemetry flow. No source leaves the machine in Local mode.
Plugin Marketplace	Capability-scoped plugins allow new tools, models or retrieval providers to be added, subject to review and signing.
Self-Healing Refactors	After applying edits, the system monitors compile errors and failing tests, automatically generates patches to fix them and iterates until success.

Key Risks & Mitigations

- **Model quality variance** – fallback and ensembling strategies mitigate poor single-model performance; continuous evaluation on SWE-bench and LiveCodeBench ensures regressions are caught [2] [3] .
- **Privacy leakage** – strict policies restrict remote calls in local mode; secrets detection and redaction protect credentials; audit logs enable compliance.
- **Prompt injections** – context cleaning, allow-lists for tool calls, and LLM safety filters reduce injection risks.
- **Performance & cost** – latency budgets per workflow, speculative decoding, caching and GPU planning keep the system responsive; cost controls allow users to set budgets.
- **Large context management** – AST-based chunking and sliding windows optimize retrieval and summarization to fit within model limits [1] .

2 Competitive Feature Parity & Gaps

2.1 Feature Matrix

The following table compares AIDE's target feature set against existing tools (Cursor, Windsurf/Codeium, Bolt.new, GitHub Copilot, Continue, Cody, Tabnine, JetBrains AI, Amazon Q Developer, Replit AI). Only short phrases are used in the matrix; full descriptions are in the text.

Feature	Cursor [4]	Windsurf/ Codeium [5] [6]	Bolt.new [7] [8]	GitHub Copilot [9] [10]	Continue [11]	Cody [12]
Autocompletion (FIM)	Multi-line and infill; proprietary models [4]	**Supercomplete** (intent-based) [5]	Basic; small templates	Standard (powered by GPT or Claude) [9]	Standard	Context-aware across repos [12]
Inline edits / refactor	Selected region or file via "Edit" command [4]	Inline AI; smart rewrites [5]	Enhance Prompt; limited editing	Edit mode: multi-file diff; agent mode for autonomous edits [9]	Edit actions via chat	Repo-level edits via chat
Agent / autonomous mode	Agent mode handles tasks end-to-end within constraints [4]	Cascade planning and agent flows (Write Mode) [18]	None	Copilot agent can iterate, run commands, create PRs [9]	Agent mode to implement features and run tests [11]	Write tests and fix across repos
Repo-level chat	Yes; references files with @	Persistent context; memories [18]	None	Yes; chat in IDE & web [21]	Yes	Yes
Code search & embeddings	Retrieval using custom models [4]	Local index & memories [18]	None	read_file tool with limited context [10]	Code search tool	Keyword & code graph search [12]
Run / execute	Terminal commands with confirmation [4]	AI terminal & Write Mode runs tasks [18]	WebContainer environment; simple preview [8]	Terminal and tests run in agent mode [10]	Command runner tool	None (requires manual run)

Feature	Cursor [4]	Windsurf/ Codeium [5] [6]	Bolt.new [7] [8]	GitHub Copilot [9] [10]	Continue [11]	Cody [12]
Test generation & debugging	Lint and error fixes; limited tests	Explain & Fix; Write Mode tests [6] [18]	None	Test generation and debugging via agents	Tools exist	Basic
Security & privacy	Unknown	Memory rules, ignore files [6]	Basic	User can restrict context; unknown privacy	Local or cloud assistants [22]	Enterprise tier restricts repo access [12]
Pricing	Subscription	Free tier; paid for advanced features	Token-based pricing [23]	Subscription; usage-based	OSS (free)	Free & paid

2.2 Opportunity Map

- **Copy**: FIM completions with Tab/Tab Tab, local indexing and retrieval rules (Cursor, Windsurf); inline multi-file edits and agent loops (Copilot agent); persistent chat with @-references and RAG (Cursor/Cody/JetBrains AI); unit test generation (JetBrains AI, Amazon Q); refactoring agent (Tabnine).
- **Improve**: Provide AST-aware retrieval and code graphs for deeper context; unify agent roles into safe loops; offer observable context to enhance trust; support offline mode with on-device models (JetBrains AI offers local via Ollama; we will extend with multi-LLM routing). Provide multi-modal capabilities (image to code and voice pair programming) missing from most competitors.
- **Avoid**: Over-autonomy where models edit code without transparency (Copilot agent mode criticisms); disregard for privacy (Bolt pushes code to the cloud) [7] ; limited context windows; single-model lock-in.
- **Parity targets**: At minimum match GitHub Copilot agent features (autonomous multi-file refactoring, terminal commands, PR creation), Windsurf Cascade flows and local index, Tabnine's personalization and protected models, JetBrains AI's Smart Apply and offline mode, Amazon Q's slash commands, and Replit's in-browser environment.

3 User Experience & Workflows

3.1 Inline Code Completion

- **FIM & suffix/prefix completions** – Press `Tab` or `Ctrl+Space` to invoke completions. The model router selects the best LLM for the language and context. Completion suggestions are streamed character-by-character; users accept with `Tab` or cycle using `Alt+[`/`]`.
- **Context** – Surrounding lines, open files and code graph neighbours are packaged into the prompt. AST-aware chunking ensures completions align with function boundaries, improving recall and generation [1] .

- **Personalization** – The router learns from user accept/reject feedback, adjusting model scores over time; per-user caches speed up repeated completions.

3.2 Command-Based Edits

1. **Invoke**: Select code or specify scope (file/directory) and trigger with `Ctrl+K` → "Edit" or type `/edit` in chat.
2. **Plan**: The system summarises the requested change, identifies impacted files via static analysis and call graph, and displays a multi-file diff plan.
3. **Preview**: Users can inspect each diff, collapse/expand changes, and approve or modify. The edit plan is anchored to AST nodes, enabling typed patch application.
4. **Apply**: After confirmation, the patch is applied using the patch DSL; a snapshot is stored for undo.
5. **Monitor**: A self-healing loop runs unit tests and static analysis; if failures occur, the agent generates fixes and updates the patch until tests pass or max retries reached.

3.3 Repo-Level Chat

- Persistent chat lives in a side panel; sessions are tied to branches. Users can ask questions about the codebase, generate documentation or summarise commit history. The chat uses RAG over the indexed repo and returns answers with citations to file:line segments.
- Chat memory is stored per repo and can recall past conversations (similar to Codeium's Memories). Users can pin notes or "memorise" context for subsequent tasks.
- `@` mentions allow referencing files or symbols; `@Web` performs targeted web search (controlled by privacy policy); `@LibraryName` fetches API docs (as in Cursor [4]).

3.4 Autonomous Task Mode

An agent orchestrator executes complex tasks through a sequence of roles:

1. **Planner** – decomposes the user's goal into sub-tasks, referencing the spec and code graph. Outputs a plan and tool call sequence.
2. **Coder** – writes code edits using diff planning; utilises context engine to fetch relevant snippets.
3. **Executor** – runs commands in a sandbox (install deps, compile, run tests). Commands are proposed by the model but require user approval before execution.
4. **Tester** – evaluates results using unit tests or generating new tests when needed; collects runtime traces.
5. **Critic** – analyses failures, logs and test output to identify root causes; loops back to Coder if fixes are needed.
6. **Reviewer** – formats the final diff, adds descriptions and citations, and prepares a Pull Request for human review.

Users can inspect and modify plans at each stage. All tool calls and code changes are visible and reversible. This workflow draws inspiration from Copilot agent mode [10], Windsurf Cascade Write Mode [18] and Continue's agent flows [11] but adds safety loops and deeper context.

3.5 Debugging Agent

- **Breakpoints & stack traces** – The IDE integrates with language runtimes to capture exceptions, variable values and call stacks. Breakpoints can be set in code; when hit, a debugging agent analyses the stack, inspects variables and suggests fixes.
- **Time-travel debugging** – Execution snapshots allow stepping backward and forward; the agent correlates variable mutations with code changes to highlight root causes.
- **Root-cause analysis** – The context engine fetches related code from the symbol graph; the agent summarises the probable cause and proposes code patches.
- **Safe changes** – Proposed patches go through the same plan/preview/apply cycle with test runs.

3.6 Code-Aware Terminal

- Terminal commands are executed in a sandbox with restricted permissions. When the user issues a command (e.g., `npm install`, `pytest`), the command runner tool checks an allowlist; if the command is risky (e.g., `rm -rf`), it requires explicit confirmation.
- The agent can suggest commands but must ask for user approval before running them. Execution output is streamed back into chat and stored in context for subsequent steps.
- A dry-run mode executes commands with `--dry-run` when available or uses simulation to preview side effects.

3.7 Spec-to-Repo Flow

- Users can supply a high-level specification (via plain text, Markdown or diagrams). The planner agent uses the context engine to infer architecture (models, controllers, APIs) and generates scaffolding with directory structure, boilerplate code, and tests.
- The code is created in a scratch branch and can be iterated via chat (refine modules, add endpoints). Integration with CI triggers builds and tests on a remote runner.

3.8 Multi-Modal Inputs

- **Diagram/image-to-code** – upload UML diagrams or UI mockups; the agent extracts components using OCR/vision models and generates corresponding code (HTML/CSS/React, DB schemas, etc.).
- **API Schema ingestion** – upload OpenAPI/GraphQL schemas and generate server/client stubs, tests and docs.
- **Voice pair-programming** – speech-to-text (STT) transcribes voice commands; text-to-speech (TTS) reads back responses. Conversations are recorded with replayable reasoning traces.

3.9 Onboarding & Accessibility

- **First-run indexing** – the indexer scans the repo, builds embeddings and symbol graphs incrementally; status is displayed. Users can select directories to exclude.
- **Privacy toggles** – choose Local, Hybrid or Cloud mode; configure allowed models and external API calls; set memory retention periods.
- **Quick wins** – tutorial tasks demonstrate completion, edit, chat and agent flows with sample projects.
- **Accessibility** – keyboard-first interactions, screen-reader friendly labels, dark/light themes, adjustable font sizes, Right-to-Left (RTL) layout support, and multi-lingual UI.

4 System Architecture

4.1 High-Level Diagram

See Figure 1 in the executive summary.

4.2 Component Breakdown

Model Router & Policy Engine (Section 6) – selects the best model per task using a scoring function, manages budgets, privacy and fallbacks.

Context Engine – comprises:

- **Embedding pipeline** – uses pre-trained code embedding models to generate vector representations of files, functions and symbols. Caches embeddings and updates incrementally when files change.
- **Code graph** – builds an abstract syntax tree (AST) and call graph for each file using Tree-sitter. Maintains cross-references and ownership metadata.
- **Retrieval** – hybrid retrieval combining BM25 lexical search and dense similarity; uses AST-aware chunking (cAST) which improves retrieval and generation quality by splitting code along syntactic boundaries [1].
- **Session memory** – stores conversation history and previous tasks; redacts secrets (API keys, passwords).

Indexer – watches filesystem via watchman; triggers incremental updates to embeddings and code graph. Supports monorepo sharding and concurrency to reduce latency.

Vector Store – pluggable back-ends: FAISS for local; PGVector or SQLite-vss for integrated persistence; Milvus/Weaviate for distributed scenarios. Implements eviction and compaction policies based on LRU and staleness.

Execution Sandboxes – local runtimes per language (virtualenv, npm/pnpm, poetry); containerized options (Docker/Podman) with seccomp/apparmor restrictions; remote VMs for heavy tasks. Snapshots allow time-travel debugging.

Agent Runtime & Tools – gRPC/HTTP microservices implement tools (file I/O, search, test runner, command runner, HTTP client, DB client). Tools return structured responses that the agent pipeline consumes.

Event Bus – streams tool calls, agent events and UI updates via gRPC or HTTP; allows UI to subscribe and update in real time.

Secrets & Policy Store – holds API keys, tokens and user preferences encrypted; enforces allowlist/denylist for external calls and shell commands.

Caching – caches prompts and model responses (Redis/SQLite) to reduce latency; caches embeddings and diff computations.

4.3 Data Flow Diagrams

Inline Completion

```
sequenceDiagram
    participant User as Developer
    participant Editor
    participant Router
    participant Context as Context Engine
    participant Model
    User->>Editor: Types code
    Editor->>Context: Request surrounding context
    Context->>Editor: AST chunks & embeddings
    Editor->>Router: Completion request (task=completion)
    Router->>Model: Select best LLM and send prompt
    Model-->>Router: Streamed tokens
    Router-->>Editor: Streamed completions
    Editor-->>User: Show inline suggestions
```

Multi-File Edit Plan & Apply

```
sequenceDiagram
    participant User
    participant Editor
    participant Planner as Planner Agent
    participant Context
    participant Coder as Coder Agent
    participant Tester as Tester Agent
    participant Sandbox
    User->>Editor: “Refactor auth logic across modules”
    Editor->>Planner: Task with context
    Planner->>Context: Query code graph & embeddings
    Context-->>Planner: Relevant files & call graph
    Planner-->>Editor: Edit plan (diff previews)
    User->>Editor: Approve plan
    Editor->>Coder: Execute plan
    Coder->>Sandbox: Apply diffs in sandbox branch
    Sandbox-->>Tester: Run tests
    Tester-->>Coder: Test results
    Coder-->>Editor: Success or repair suggestions
    Editor-->>User: Show final diffs and results
```

Agentic Pull Request Workflow

```
flowchart LR
    A(Start) --> B{Task}
    B -->|Simple edit| E[Inline Edit Mode]
    B -->|Complex| C[Planner Agent]
    C --> D[Coder Agent]
    D --> F[Executor Agent]
    F --> G[Tester Agent]
    G --> H[Critic Agent]
    H --> I{Success?}
    I -->|No| D
    I -->|Yes| J[Reviewer Agent]
    J --> K[Create Pull Request]
```

Debug Loop

```
sequenceDiagram
    participant User
    participant DebuggerUI
    participant DebugAgent
    participant Context
    participant Sandbox
    User->>DebuggerUI: Set breakpoint & run
    Sandbox-->>DebuggerUI: Breakpoint hit (stack trace)
    DebuggerUI->>DebugAgent: Send stack & variables
    DebugAgent->>Context: Retrieve related code
    Context-->>DebugAgent: Snippets & call graph
    DebugAgent-->>DebuggerUI: Root-cause analysis & suggestions
    User->>DebuggerUI: Accept fix
    DebuggerUI->>Coder: Apply patch via agentic flow
```

5 Language Support & LSP Integration

Supported Languages

AIDE aims to cover the top 20 languages by GitHub popularity: Python, JavaScript/TypeScript, Java, C/C++, C#, Go, Rust, PHP, Ruby, Swift, Kotlin, Shell, SQL, R, Scala, Haskell, Dart, Lua, Clojure, and Julia. Support includes syntax highlighting, LSP features (hover, diagnostics, completions) and integration with language-specific tooling.

Adapters

For each language, AIDE implements an adapter that wraps the language server (e.g., Pyright, tsserver, gopls). The adapter extends LSP with new methods:

- `workspace/editPlan`: returns multi-file diff plans for a natural language instruction.
- `workspace/applyPatch`: applies a typed patch with rollback support.
- `workspace/getSymbolGraph`: retrieves call graph and symbol relationships for retrieval.
- `textDocument/runTests`: runs tests for the current file or project.

Adapters provide AST transformers to map tree-sitter ASTs to our internal representation and supply refactoring primitives (rename, extract method, convert loop, etc.). Each language also has test scaffolding templates (pytest/unittest for Python, JUnit/TestNG for Java, etc.) and runtime adapters to run code in sandboxes.

6 LLM Strategy: Auto-Selection + Overrides

6.1 Model Profiles (Appendix summarises details)

Model	Context window	License	Hosting options	Observed strengths	Weaknesses
Codestral 25.01	256k tokens [25]	Mistral AI Non-Production license with commercial options [26]	API (Mistral, Vertex AI) or self-hosted; local via GGUF/ quantization	SOTA on FIM tasks and HumanEval (86.6% pass@1) [27]; fast with improved tokenizer [28]	Requires 22B parameters and GPU; license restricts commercial use without contract.
Code Llama 70B Instruct	16k trained, extended to 100k tokens via RoPE fine-tuning [29] [30]	Permissive license allowing commercial use [31]	Download via Hugging Face; run locally or via service providers	Strong open-source baseline; supports FIM; high HumanEval/ MBPP scores (67%/65%) [32]	70B model requires large GPUs; slower inference; limited to 100k context.

Model	Context window	License	Hosting options	Observed strengths	Weaknesses
DeepSeek Coder V3	128k context (V3.1) [33]	Custom commercial license (DeepSeek) with restrictions	API or self-hosted (H200/A100 GPUs)	Hybrid reasoning architecture improves programming tests; cost-effective (71.6% Aider pass, 68× cheaper than Claude Opus) [34]	Very large model (671B total parameters with 37B active); may require FP8; slower response times [34] .
Qwen 2.5 Coder (7B/14B/32B)	128k context for 7B/14B/32B variants; 32k for smaller models [35]	Apache 2.0 license for 0.5B/1.5B/7B/14B/32B; Qwen Research license for 3B [35]	Available via Hugging Face and ModelScope; local or API	SOTA open-source model; strong code repair and reasoning; multi-language support across 40+ languages [36] ; 32B instruct matches GPT-4o for coding [37]	Slightly smaller context (128k) than Codestral; 32B model heavy; licensing of 3B variant restricts commercial use.
Llama 3.1 (8B/70B/405B)	128k context [38]	Custom license (commercial use allowed; can use outputs to improve other models) [39]	Available on Hugging Face and major cloud providers	405B model rivals top closed models; strong general knowledge and math reasoning; 8B/70B instruct tuned; supports tool use and eight languages [40]	405B model requires enormous compute; 8B/70B may lag behind code-specific models on programming tasks; license applies to derivatives.

6.2 Routing Policy

The model router ranks models by computing a **score** for each candidate using the following factors:

- **Task type** (`completion`, `edit`, `debug`, `test-gen`, `planning`, `math`) – weights based on benchmark rankings (e.g., Codestral excels at FIM; Qwen at repair; Llama 3.1 at logic).
- **Languages** – some models specialise (e.g., Code Llama for Python, Qwen for many languages). Weighted by language mapping table.
- **Context length required** – estimated prompt length relative to model capacity; penalise models with insufficient context.
- **Latency budget** – user preferences (low latency vs high quality); approximate model speed/cost factors from observed latencies.
- **Privacy mode** – local-only tasks restrict to on-device models (e.g., Code Llama, Qwen) or offline engines; hybrid/cloud mode includes remote APIs.
- **Model quality** – baseline performance scores per task from evaluations (HumanEval, LiveCodeBench, SWE-bench). Updated regularly.
- **User overrides** – explicit preferences override scores.

Scoring Function (Pseudo-Code)

```
def select_model(task: str, language: str, prompt_len: int, mode: str,
latency_pref: str, overrides: list) -> str:
    # Predefined model profiles with scores per task
    models = ["codestral_25_01", "codellama_70B", "qwen25_coder_7B",
"qwen25_coder_32B", "llama31_70B", "deepseek_v3"]
    scores = {}
    for m in models:
        if m in overrides.allow_list:  # optional allow/deny lists
            score = base_score[m][task]
            score *= language_weight[m].get(language, 0.8)
            if prompt_len > context_limit[m]:
                score *= 0.1  # penalise overflow
            if mode == "local" and not is_local(m):
                score = 0
            if latency_pref == "latency":
                score *= 1 / latency_estimate[m]
            else:
                score *= quality_estimate[m]
            scores[m] = score
    # Sort and select highest
    ranked = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return ranked[0][0] if ranked else None
```

The router caches the last selection per user and updates model metrics based on evaluations. Fallbacks: if a model times out, choose the next highest; partial completions may be merged by combining outputs from multiple models using a voting/critic mechanism.

6.3 User Overrides & Policies

Users can set global preferences (e.g., “prefer open models,” “latency first”) or per-task overrides (choose Qwen for code repair). Policies can be encoded in YAML/JSON with sections for budgets, privacy, allowed tools and allowed models. Example in Appendix.

6.4 Safety Gates

Routing integrates with the policy engine to prevent suggestions that violate user settings (e.g., running commands that modify global system files). For shell/tool execution, the router consults a risk matrix (Section 10) and ensures models cannot propose high-risk commands without gating.

6.5 Prompt Templates & Context Packer

For each model and task, we maintain prompt templates with placeholders for instruction, code snippets, context and constraints. A **context packer** builds a condensed representation: (1) summarises open files, (2) extracts AST nodes, call graphs and failing tests, (3) deduplicates overlapping text, (4) redacts secrets. When prompt length exceeds the model’s context, the packer applies sliding windows and recency weighting. Summarization strategies include map-reduce and AST summarisation.

7 Context & Knowledge Systems

7.1 Code Graph

- **Symbol table** – maps symbols (functions, classes, variables) to definitions and references.
- **Call graph** – directed graph of function/method calls; includes dynamic calls resolved via heuristics.
- **Dependency graph** – module/package dependencies (imports, includes). Captures third-party libraries and version constraints.
- **Ownership map** – tracks ownership (author, owner file) to prioritise retrieval (owner heuristics for error localisation).

Graphs are updated incrementally; modifications trigger local recomputation. Graph queries assist the planner and retrieval functions.

7.2 Retrieval-Augmented Generation (RAG) for Code

RAG is essential for long-range reasoning. We adopt a hybrid retrieval strategy:

- **Lexical search** using BM25 or code-aware tokenization to quickly find files by keywords.
- **Dense embeddings** using pre-trained code embedding models; retrieval by nearest neighbours.
- **AST-aware chunking** (cAST) splits code into syntactic units rather than arbitrary lines, improving retrieval and generation quality [1] .
- **Ranking** – combine lexical and semantic scores; adjust by locality (files close in dependency graph), recency (recently modified files) and staleness (older references decay over time).

The retrieval engine returns a set of context packets (snippets with metadata). Each packet includes the file path, start/end lines, symbol names and embeddings ID. Citations reference these packets.

7.3 Long-Context Strategies

Models with large context windows (Codestral 256k, Qwen 128k, Llama 3.1 128k) can process substantial input, but we still need policies to prioritise relevant content. Strategies:

- **Sliding windows** – for iterative tasks, shift the window across large files, summarising previous sections to maintain continuity.
- **Context packets** – group retrieved snippets into self-contained packets; use hierarchical summarisation when overflows occur.
- **Recency weighting** – assign higher scores to recently executed code or conversation items.
- **Deduplication & compression** – remove duplicate lines and compress unchanged code; use diff representation for multi-file edits.

7.4 Memory & Redaction

Memory is stored per repo/branch and per user. It includes chat history, agent actions, retrieved context and test results. Secrets (API keys, tokens, passwords) are detected via regex and heuristics and replaced with tokens (e.g., `[SECRET]`). Redaction policies enforce that these tokens are not sent to remote models in local mode.

8 Agentic Workflow Design

8.1 Agent Roles & Tools

Agents are implemented as LLM prompts combined with tool call functions. Roles:

1. **Planner** – uses natural language planning prompts to break tasks into steps. Tools: `code_search`, `symbol_graph`, `doc_lookup`.
2. **Coder** – uses `write_file`, `diff_tool`, `apply_patch`, `search_code` to implement steps. Accepts diff suggestions from the model but always uses typed patch DSL for safety.
3. **Executor** – uses `run_command` tool to execute scripts/tests in sandbox; supports dry-run and simulation. Observes logs.
4. **Tester** – uses `run_tests` to execute test suites; generates additional tests via `generate_tests` using a prompt template.
5. **Critic** – analyses test failures and runtime errors; uses `search_logs` and `debug_symbol` tools to identify issues. May call the planner again.
6. **Reviewer** – summarises changes, ensures coding standards, runs linters, and prepares PR with metadata and citations. Tool: `create_pr` or `submit_patch` (depending on VCS platform).

8.2 Safety Loop

The workflow incorporates safety checks at each stage:

- **Verify** – before executing code, the agent ensures that the patch applies cleanly and does not delete critical files; static analysis identifies potential security issues.
- **Simulate** – run commands in dry-run mode; for commands lacking built-in dry-run flags, the system uses instrumentation to detect file system writes and prompts the user.

- **Apply** – commit changes only after tests pass and the user approves. A snapshot is stored for rollback. High-risk operations (database migrations, file deletions) require double confirmation.
- **Rollback** – if tests or lints fail after merge, the self-healing loop triggers; if unsolved, the system reverts to the snapshot and notifies the user.

8.3 Multi-File Patch Planning

Plan generation uses call graph analysis to determine affected files. The diff plan groups changes by file and reason (e.g., “Add new function `validate_user` in utils.py”; “Rename class `AuthManager` to `LoginManager`”). Each change is represented in a typed patch DSL that enforces AST constraints (e.g., cannot break syntax). The user preview displays file/line numbers and highlights modifications. After approval, the patch is applied via the patch engine (Section 12). The patch DSL ensures that modifications keep the AST valid; type checks run before applying.

9 IDE Platforms & Packaging

9.1 Primary Platforms

- **VS Code extension** – built with TypeScript; integrates with the LSP; uses the AIDE backend via gRPC/HTTP. Supports Windows, macOS (x86 & Apple Silicon) and Linux.
- **JetBrains plugin** – written in Kotlin; leverages the IntelliJ platform’s PSI (Program Structure Interface) and refactoring capabilities. Integrates Junie features by hooking into run configurations and code inspections. Supports offline mode via local models.

9.2 Standalone Shell

A lightweight desktop application built with Tauri/Electron for developers who prefer an independent IDE. Uses the same backend and UI components; leverages WebAssembly for performance-critical modules (e.g., embeddings, search). The shell includes built-in version control and integrated terminal.

9.3 Cross-Platform Considerations

- Provide prebuilt binaries for major OS/architectures; support GPU acceleration for on-device models (CUDA on Linux/Windows, Metal on macOS). Use quantized models (AWQ, GPTQ) where possible.
- Concurrency: background services run as isolates; watchers debounced to avoid thrashing; asynchronous indexing and prefetch.
- Support remote development (e.g., SSH, Codespaces) by running the backend in the remote host and forwarding UI events.

10 Privacy, Security, Compliance & Safety

10.1 Privacy Modes

- **Local-only** – all processing occurs on the local machine; only local models are used; no telemetry or external calls; secrets never leave the device. Ideal for air-gapped environments.
- **Hybrid** – retrieval and context remain local; model calls may go to selected remote providers (e.g., Mistral or Qwen) via encrypted channels. Telemetry is hashed and aggregated.

- **Cloud** – full access to remote models; tasks may call search and knowledge APIs. Telemetry includes anonymized usage metrics.

Users choose a mode at setup and can switch per project. Policies enforce restrictions for each mode.

10.2 Data Governance

- Source code and embeddings are stored encrypted on disk. In local mode, nothing is uploaded. In hybrid/cloud, redacted context and hashed telemetry may be sent. Users can set retention periods and request deletion.
- Telemetry includes latency, success rates, model usage and anonymised error codes. Sensitive data (file content, variable names) is not logged.

10.3 Secrets Handling

- The secrets scanner uses regex patterns and machine learning to detect API keys, passwords, tokens, private keys, and credentials. When detected, secrets are replaced with `[SECRET]` tokens before context is sent to models. The actual secrets are stored in the secure vault and accessible to tools only when needed (e.g., `run_command` uses environment variables). JetBrains AI offers similar vault integration [14]; we extend this across AIDE.

10.4 Supply-Chain Safety

- The dependency auditor uses SBOM (Software Bill of Materials) to list dependencies and their licenses, scanning for vulnerabilities using SCA/SEA tools. In agent mode, suggestions to install packages are checked against allowlists and license requirements.
- Third-party dependencies in plugins are signed and verified. The plugin marketplace has a review process to detect malicious code.

10.5 LLM Safety & Prompt Injection Defences

- The system employs prompt filters to strip injection attempts (e.g., `## Prompt injection instructions`) and restrict instructions that request secrets or break tasks.
- Output filters remove code that includes suspicious patterns (e.g., reading `/etc/passwd`) or high-risk commands.
- The router's safety gate denies tool calls flagged as high-risk by the policy engine, requiring manual confirmation. For instance, commands like `rm -rf` or `DROP DATABASE` require explicit user approval. A risk matrix categorises commands into safe, moderate and dangerous.

10.6 Compliance & Security

- Pursue SOC2 Type II and ISO/IEC 27001 compliance. Features: SSO (OIDC/SAML), audit logs capturing tool calls and model responses, data residency controls.
- Implement GDPR/CCPA mechanisms: data access requests, deletion, purpose limitation.
- Enterprise SSO ensures that user identities are integrated with company policies; roles restrict capabilities (e.g., interns cannot run `deploy`).

11 Extensibility & Plugin Model

11.1 Extension API

Plugins can extend the IDE with new commands, tools, providers or renderers. Each plugin declares a manifest specifying:

- **Capabilities** – e.g., `code_search_provider`, `test_runner`, `language_model`.
- **Permissions** – resources it can access (file system, network, commands); scope is restricted by capability.
- **Versioning** – semantic version; compatibility with core API versions.

Plugins use gRPC/HTTP interfaces and adhere to JSON/gRPC schemas defined in the developer kit. Plugins run in isolated sandboxes to prevent unauthorized access.

11.2 Model Provider SDK

Providers can register new LLMs by implementing an adapter interface: `init()`, `call(prompt, stream)`, `metadata()`. The adapter returns its context limit, licensing info, and cost profile. Providers may implement local inference or remote API calls. The router discovers providers via a capability registry and uses metadata for scoring.

11.3 Tool Marketplace

The marketplace distributes plugins. Submissions undergo automated security scanning and manual review. Packages are signed; the client verifies signatures before installation. A rating system and metadata (compatibility, categories) help users discover tools.

11.4 Schema & Sample Plugin

The API defines message schemas for tool calls (e.g., `RunCommandRequest`, `SearchCodeRequest`, `CreatePRRequest`). Example plugin: `grep` code search provider uses ripgrep to search the workspace; implements `code_search` capability and returns file paths and line numbers.

12 Execution & Sandboxing

12.1 Local Runners

- For each language, we spin up isolated environments (virtualenv for Python, Node.js `npm/pnpm`, Go modules) with resource caps (CPU, memory). The sandbox restricts file system access to the project directory, prohibits network access by default and uses seccomp or AppArmor on Linux.
- **Dev Containers** – optional containerised environments with pinned toolchains; integrated with VS Code Dev Containers specification.

12.2 Remote Runners

- For compute-heavy tasks or GPU inference, remote sandboxes are provisioned on demand. Each sandbox uses an immutable base image; tasks are executed, logs captured and results returned. Snapshots allow time-travel debugging and reproduction of failing tests.
- Integration with CI systems (GitHub Actions, GitLab CI, Jenkins) allows reproducing test failures and running pipeline tasks through the agent. The agent can fetch logs, interpret test results and propose fixes.

12.3 Determinism & Hermetic Builds

- Enforce pinned dependencies via lockfiles (requirements.txt, package-lock.json). The agent never auto-updates dependencies unless the plan includes it. Hermetic builds (e.g., Nix) are supported to guarantee reproducibility.

13 Telemetry, Observability & LLMOps

13.1 Metrics

- Latency per action (completions, edits, test runs), token usage, cost per model, success rate (tasks completed vs failed), hallucination flag rate, test pass rates, revert rates.
- Agent loop metrics: number of iterations, tool call counts, coverage of tests, mean time to resolution.

13.2 Tracing

- Use OpenTelemetry to instrument tool calls, agent flows and model interactions. Each prompt/response is tagged with session IDs, user IDs, model version and context packet. Traces allow replay for debugging and accountability.

13.3 Evaluation & Benchmarks

- Continuous evaluation harness runs nightly on curated tasks covering multiple languages and frameworks (SWE-bench tasks for real-world bug fixes [41], LiveCodeBench tasks for holistic evaluation [42], HumanEval/MBPP subsets, LiveCodeBench Live). A blend of closed and open models is used to evaluate responses.
- **LLM-as-judge** – to reduce manual evaluations, we use LLMs (e.g., GPT-4o) as judges for code quality; human spot checks ensure accuracy. Metrics: pass@1, run time, style adherence.
- AB testing of prompts, routing strategies and context packing; gating criteria define when a new strategy can roll out.

14 Performance Engineering

14.1 Latency & Throughput Budgets

- Inline completion: target <200 ms average latency (similar to Ghostwriter's <400 ms median [17]). Achieved via streaming, model caching and speculative decoding.
- Multi-file edits: plan generation in <2 s for typical tasks; diff application and test runs <10 s; agent loops should complete within user-set budgets (default 5 min for complex tasks).

- Chat responses: <1 s for RAG retrieval; <5 s for model answer.

14.2 Token & Memory Management

- Deduplicate context; compress repeated sequences; reuse embeddings across tasks.
- Manage KV-cache for local models; share caches across streams.
- Use quantized models (AWQ, GPTQ) to run 70B models on single GPUs; share weights across sessions via memory mapping.

14.3 Cost Models & Tiers

- Provide performance tiers: **Local** (offline; limited to open models; free), **Standard** (hybrid; moderate cost; uses open models and smaller remote models), **Pro** (full cloud; premium models like Codestral 25.01 and DeepSeek V3). Costs are estimated per token and per workflow; budgets can be set in policies.

15 Roadmap & Delivery Plan

15.1 Phases

1. **MVP (Months 0-3)** – Build core UI for VS Code extension; integrate at least one open model (Code Llama 70B) locally; implement indexer and embeddings; deliver inline completions, chat and simple edits. Provide local mode only.
2. **Beta (Months 3-6)** – Add agentic workflows (planner, coder, executor); integrate remote models (Codestral, Qwen) via router; implement vector store; support test runs and debugging; release JetBrains plugin; roll out plugin marketplace in closed beta.
3. **GA (Months 6-12)** – Full agent loop with Reviewer; integrate privacy modes; open plugin marketplace; add multi-modal inputs; deliver self-healing refactors and typed patch DSL; achieve parity with Copilot agent and Windsurf Cascade.

15.2 Staffing & Teams

- **Core architecture**: LLM integration engineers (routing & models), context/retrieval engineers, agent framework engineers, UI/UX designers, security engineers.
- **Plugin & ecosystem**: Developer relations, plugin marketplace curator, API designers.
- **Compliance & Security**: Legal counsel, compliance officer, security analysts.
- **Product & QA**: Product managers, QA engineers, evaluation specialists.

15.3 Risks & Dependencies

- Availability of high-quality open models with large contexts; potential licensing changes.
- Editor API stability (VS Code, JetBrains); OS updates.
- GPU supply for local deployments; performance of quantized models.
- User adoption and trust; need to demonstrate value over existing tools.

16 Detailed Specifications & Artifacts

16.1 Architecture Diagrams

Diagrams are provided throughout this document (Figures 1–4). Additional diagrams can be exported to documentation.

16.2 Model Router Artifact

- **Pseudo-code**: Provided in Section 6.2.
- **Config Schema**: Example (YAML)

```yaml
model_routing:
  default_policy: quality_first
  budgets:
    latency_ms: 5000
    cost_tokens: 20000
  privacy_mode: hybrid
  models:
    codestral_25_01:
      allowed: true
      max_tokens: 256000
      cost_per_million_tokens: 8.00
    codellama_70b:
      allowed: true
      max_tokens: 100000
      cost_per_million_tokens: 0.00  # local
    qwen25_coder_7b:
      allowed: true
      max_tokens: 128000
      cost_per_million_tokens: 0.00  # open local
    qwen25_coder_32b:
      allowed: true
      max_tokens: 128000
      cost_per_million_tokens: 2.00
    llama31_70b:
      allowed: true
      max_tokens: 128000
      cost_per_million_tokens: 1.50
  allow_list: []
  deny_list: []
```

16.3 Context Engine Data Schemas

```
// Embedding record
{
  "id": "UUID",
  "file": "src/utils/auth.py",
  "span": { "start_line": 10, "end_line": 40 },
  "tokens": [123, 45, ...],
  "vector": [0.23, -0.1, ...],
  "symbol_names": ["validate_user", "AuthError"],
  "last_modified": "2025-08-23T17:00:00Z"
}

// Citation anchor
{
  "anchor_id": "UUID",
  "file": "src/auth.py",
  "line_range": [50, 55],
  "context_id": "UUID"
}
```

16.4 Ranking Formula

The final retrieval score is:

$$score = \alpha \cdot \mathrm{BM25}(q, d) + \beta \cdot \mathrm{cosine}(\vec{q}, \vec{d}) + \gamma \cdot \mathrm{locality}(d) + \delta \cdot \mathrm{recency}(d) - \epsilon \cdot \mathrm{staleness}(d),$$

where $\alpha, \beta, \gamma, \delta, \epsilon$ are tunable weights; locality is based on call graph distance; staleness decays older contexts.

16.5 Prompt Templates

Each template is parameterised with variables ({instruction}, {code}, {context}, {constraints}). Example for FIM completion:

```
You are a coding assistant. Complete the code between the tags:
<prefix>
{prefix}
</prefix>
<suffix>
{suffix}
</suffix>
Use the context below if necessary:
{context}
Return only the code for the missing part.
```

Similar templates exist for edit-diff, planning, critic, reviewer, debugger and test generation. Guardrails specify language, style, tests and safety instructions.

16.6 APIs & Interfaces

Define gRPC/HTTP services:

- `ModelProvider` – `Call`, `Metadata`.
- `Tool` – `CallTool` with tool-specific request/response; capability enumeration.
- `Agent` – `StartTask`, `GetStatus`, `StopTask`.
- `Indexer` – `IndexRepo`, `GetEmbedding`, `GetSymbolGraph`.
- `ExecutionRunner` – `RunCommand`, `RunTests`, `ExecuteTask`.

LSP extensions: new methods `workspace/editPlan` and `workspace/applyPatch` for multi-file diff planning and application.

16.7 Security & Safety Artifacts

- **Threat model** – identify threat actors (malicious plugin developer, compromised model provider, insider), assets (source code, secrets, infrastructure) and attack vectors (prompt injection, supply-chain attacks, exfiltration). Mitigations include plugin sandboxing, code signing, model audit, and network segmentation.
- **Command matrix** – classify commands into safe (ls, cat), moderate (npm install), high-risk (rm -rf, drop database). Implementation ensures high-risk commands require triple confirmation.
- **Redaction policies** – specify patterns and heuristics; maintain allowlist for non-sensitive tokens.

16.8 UX Wireflows

Provide wireflow diagrams (not included here due to space) for: inline completion, edit preview/apply, agent plan review, terminal agent, debug assistant and PR creation. Use consistent keybindings: `Ctrl+K` for AI actions, `Ctrl+Shift+A` to open agent panel, `Alt+Enter` to accept suggestions. Discoverability: tooltips, command palette entries, first-run tour.

16.9 Testing & Evaluation Plans

- **Unit tests** – cover core logic (routing, indexing, patch DSL). Use property-based tests for patch validation.
- **Integration tests** – simulate editing flows across languages; test agent loops on sample repos.
- **End-to-end** – run full tasks from spec to PR on open-source projects; measure latency and success.
- **Benchmarks** – nightly runs on SWE-bench Lite (300 tasks) [41], LiveCodeBench tasks [42], and synthetic tasks; log pass rates.
- **CI/CD** – integrate with GitHub Actions; break-glass procedures for major failures; auto-rollback of plugin marketplace updates.

16.10 Performance Budgets

- Document memory ceilings per model (see model table); specify maximum concurrent tasks (e.g., 5 agents per CPU). Offline mode budgets: <8 GB RAM usage for local models; <2 GB for context index; avoid GPU swap. Provide guidelines for scaling: number of models loaded vs available VRAM.

16.11 Extensibility Guide

Instructions for adding a new LLM:

1. Implement the adapter in `model_providers/<name>.py` with `metadata`, `call` and `load_model` methods.
2. Register the model in the configuration with context limits, cost and license.
3. Add evaluation results to the routing table; update weights.
4. Write unit tests verifying integration and fallback behaviour.

Instructions for adding a new tool:

1. Define a gRPC service with request and response messages.
2. Implement the tool server; handle input validation and error propagation.
3. Register capability in plugin manifest; assign permissions.
4. Provide a front-end UI component if needed (e.g., file picker).

17 Novel Innovations Beyond State-of-the-Art

1. **Intent-to-Change Graph** – Represent planned edits as a graph of AST nodes, linking intentions to impacted code. This structure allows static analysis to predict potential side effects and provides a visual diff for users. The planner uses the graph to ensure comprehensive refactoring and avoid missing related changes.

2. **Observable Context Packets** – Expose the exact snippets and metadata sent to the model in a side panel. Users can inspect and redact content before sending. This transparency builds trust and aids debugging of prompt issues.

3. **Self-Healing Refactors** – After applying a patch, a monitor watches compilation/test results. If failures occur, the agent automatically generates repair patches using retrieval over failing code and error messages; it iterates until the project builds. This is inspired by LLM-based program repair research [43].

4. **Temporal Code Memory** – Maintain recency-weighted embeddings; each context packet has a “stability score” that decays over time. Retrieval favours recent code unless older code is explicitly referenced. This addresses long-term drift in large repositories.

5. **Constraint-Driven Planner** – Let users specify constraints (latency < 2 s, context $\leq$ 50k tokens, cost < $1) and preferences (prefer open models). The planner uses these constraints to choose models, summarisation strategies and test budgets. Exposes a slider UI for cost/quality trade-offs.

6. **Live Design Mode** – Generate architecture decision records (ADRs) and maintain diagrams as the code evolves. When adding a new component, the system proposes updates to UML/ADR documents and ensures they stay synchronised with the code.

7. **Spec-to-CI** – From a spec, derive not only code but also CI pipelines, data fixtures and environment definitions (Dockerfiles, Kubernetes manifests). The system keeps CI scripts in sync with code changes and tests.

8. **Typed-Patch DSL** – A domain-specific language for expressing multi-file patches with type annotations and invariants. The patch engine statically ensures that patch operations preserve syntax and types; misaligned patches are rejected before application.

9. **Runtime-Aware Suggestions** – Integrate profiling data (CPU usage, memory, I/O) into context; suggest performance optimisations and highlight bottlenecks. The agent can plan micro-benchmarks and propose refactors (e.g., replace nested loops with vectorised operations).

10. **Risk-Scored Suggestions** – Each AI suggestion includes a risk score and explanation (e.g., high risk of security vulnerability, potential breaking change). The user can filter by risk tolerance and ask the agent to propose safer alternatives.

Appendices

A Model Profiles

This appendix summarises the specified models (Codestral 25.01, Code Llama 70B, DeepSeek Coder V3, Qwen 2.5 Coder 7B/14B/32B, Llama 3.1 8B/70B/405B). For each model we include context window, licensing, hosting options, strengths, weaknesses and benchmark standings. See Section 6.1 for the table.

B Example Routing Policy

Provided in Section 16.2.

C Mermaid Snippets

The diagrams used in this specification are included as code blocks. They can be embedded directly in documentation.

D Glossary

- **AST** – Abstract Syntax Tree, hierarchical representation of code structure.
- **RAG** – Retrieval-Augmented Generation; technique of augmenting LLM prompts with retrieved documents.
- **FIM** – Fill-In-the-Middle; code completion task where the model predicts content inside a hole between prefix and suffix. [44]
- **SWE-bench** – benchmark of 2,294 real GitHub issues requiring multi-file code changes [41] .
- **LiveCodeBench** – continuous benchmark collecting coding problems across time; evaluates code generation, self-repair and test output prediction [3] .
- **LLM** – Large Language Model.

E Open Questions & Experiments

1. **Model performance on huge monorepos** – Evaluate retrieval strategies and context packing on repositories with >10k files. Investigate sliding windows and summarisation heuristics.
2. **Offline compression** – Research incremental compression of code embeddings to reduce disk footprint in local mode.
3. **Agent decision transparency** – Study UI designs for representing the agent's reasoning steps; measure user trust and intervention rates.
4. **User-driven evaluation** – Develop simple interfaces for users to rate suggestions; feed back into model scoring and prompt design.
5. **Integration with enterprise ecosystems** – Explore connectors to internal issue trackers, knowledge bases and CI/CD pipelines, respecting data governance.

---

[1] cAST: Enhancing Code Retrieval-Augmented Generation with Structural Chunking via Abstract Syntax Tree
https://arxiv.org/html/2506.15655v1

[2] [41] SWE-bench: Can Language Models Resolve Real-world Github Issues? | OpenReview
https://openreview.net/forum

[3] [42] LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code
https://livecodebench.github.io/

[4] Features | Cursor - The AI Code Editor
https://cursor.com/features

[5] [18] Windsurf AI Agentic Code Editor: Features, Setup, and Use Cases | DataCamp
https://www.datacamp.com/tutorial/windsurf-ai-agentic-code-editor

[6] Windsurf - Cascade
https://docs.windsurf.com/windsurf/cascade/cascade

[7] Bolt.new vs Cursor vs Replit Agent: Which AI coding tool is best?
https://refined.so/blog/bolt-cursor-replit-comparison

[8] [23] What is Bolt.new AI: Comprehensive Review by A Pro Vibe Coder
https://www.prismetric.com/what-is-bolt-ai/

[9] [21] GitHub Copilot features - GitHub Docs
https://docs.github.com/en/copilot/get-started/features

[10] Introducing GitHub Copilot agent mode (preview)
https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode

[11] Quick Start - Continue
https://docs.continue.dev/features/agent/quick-start

[12] Cody vs. Replit AI: Comparing AI Code Generation tools (2025) | Greptile Blog
https://www.greptile.com/blog/comparing-cody-vs-replit

[13] AI code refactoring: 7 ways Tabnine transforms refactoring - Tabnine
https://www.tabnine.com/blog/ai-code-refactoring-7-ways-tabnine-transforms-refactoring/

[14] AI Assistant Features
https://www.jetbrains.com/ai-assistant/

[15] [24] Junie, the AI coding agent by JetBrains
https://www.jetbrains.com/junie/

[16] Hands-On Amazon Q Developer Latest Features - /dev, /review, /doc, /test, /transform - DEV Community
https://dev.to/aws-builders/hands-on-amazon-q-developer-latest-features-dev-review-doc-test-transform-1m9l

[17] Replit — Ghostwriter AI & Complete Code Beta
https://blog.replit.com/ai

[19] Unveiling Tabnine's Code Review Agent: Improving quality, security, and compliance uniquely for every development team - Tabnine
https://www.tabnine.com/blog/unveiling-tabnines-code-review-agent/

[20] JetBrains details deeper mechanics of Junie coding agent - Techzine Global
https://www.techzine.eu/blogs/applications/133356/jetbrains-details-deeper-mechanics-of-junie-coding-agent/

[22] Introduction - Continue
https://docs.continue.dev/

[25] [27] [28] [44] Codestral 25.01 | Mistral AI
https://mistral.ai/news/codestral-2501

[26] Codestral | Mistral AI
https://mistral.ai/news/codestral

[29] [30] [31] [32] Code Llama: Open Foundation Models for Code
https://arxiv.org/html/2308.12950v3

[33] [34] DeepSeek V3.1 Complete Evaluation Analysis: The New AI Programming Benchmark for 2025 - DEV Community
https://dev.to/czmilo/deepseek-v31-complete-evaluation-analysis-the-new-ai-programming-benchmark-for-2025-58jc

[35] [36] [37] Qwen2.5-Coder Series: Powerful, Diverse, Practical. | Qwen
https://qwenlm.github.io/blog/qwen2.5-coder-family/

[38] [39] [40] Introducing Llama 3.1: Our most capable models to date
https://ai.meta.com/blog/meta-llama-3-1/

[43] A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications
https://arxiv.org/html/2506.23749v1