ChatGPT

# Designing a Greenfield LLM Project with Interface Stubs and Graph Analysis

Developing a new **greenfield project** with the assistance of Large Language Models (LLMs) calls for careful planning and novel techniques. One promising approach is to **compress the code logic** into a minimal specification (~1% of the codebase) defined by interface signatures and relationships, and then let LLMs generate the actual code. By first capturing the system as a concise interface-level blueprint and **knowledge graph**, we can analyze coverage, performance budgets, idempotency, and consistency *before* writing code. Below, we **brainstorm deeper** on each idea, covering everything from interface-stub specs to graph-based analysis tools, cross-stack flow mapping, and visualization, with an eye towards open-source implementation.

## 1. Interface-Stub Architecture Specification (1% Code Blueprint)

**Idea:** Begin by designing an interface-level specification that represents the entire system's logic in a compact form (targeting ~1% of the eventual code size). This spec consists of **interface stubs** (function signatures, type definitions, trait contracts) along with structured metadata about requirements, operations, and constraints. The goal is an authoritative JSONL (JSON Lines) or IDL document that declares all **types, methods, relationships, scenarios, guards, and policies** for the project. Essentially, this is a high-level *architecture spec* capturing what the code will do without writing the implementation.

- *Contents:* Each entry in the spec can define an interface or component (with its methods and data types), or a higher-level requirement or scenario. For example, one JSON object could represent a function signature (inputs, output, purpose), and others could represent:
- **Edges** – relationships like "Function A calls Function B" or data flows between components.
- **Scenarios** – user stories or usage flows (sequences of calls representing a feature).
- **Guards/Preconditions** – conditions that must hold true (invariants, input validation rules).
- **Policies** – non-functional requirements like performance budgets (e.g., "this operation must complete < p99 200ms"), idempotency guarantees, consistency models, etc.
- *Benefits:* By sketching the system in this abstract form, we can perform **pre-mortem analysis** on the design. The spec can be reviewed to catch missing cases or conflicts, and even probed by an LLM for weaknesses. In practice, LLM-driven workflows have found it effective to *"brainstorm spec, then plan, then execute using LLM codegen"*, iteratively refining a detailed specification before coding [1] [2]. This spec serves as a single source of truth that a developer (or an LLM) can later use to generate actual code stubs and implementations.

**Feasibility:** This idea echoes known best practices. For instance, some developers use a multi-phase approach where *the LLM first generates a complete Interface Definition Language (IDL) spec for the use case, then produces code from that spec* [3]. By investing effort in a rigorous interface design up front, we effectively compress the core logic into a human- and machine-readable form. This blueprint (roughly "1% of the code") can drive all subsequent steps, helping ensure coverage of all requirements and clarifying invariants and performance budgets early. We can even have a "pre-code review" of the spec: ask a

reasoning LLM to **poke holes in the design** and surface potential issues (a sort of automated pre-mortem analysis) [4]. This way, concerns like idempotency or consistency issues can be identified at the specification stage. Overall, an interface-stub architecture spec provides a **safety net** and clarity for LLM-based development, making the code generation phase more reliable and aligned with the system's true requirements.

## 2. Minimal Code Knowledge Graph (Functions, Types, Traits & Relations)

**Idea:** Represent the codebase as a **graph** of interconnected parts – specifically a "three-by-three" graph where **nodes** are of three types (Function, Type, Trait) and **edges** capture relationships like *calls*, *interacts with*, and *implements*. In this graph, every function, data type (class/struct), and trait (interface) is a node identified perhaps by a **signature hash (SigHash)** for uniqueness. Edges then represent: - **Calls:** e.g. Function A →calls→ Function B. - **Implements:** e.g. Type X →implements→ Trait Y (or a class implementing an interface). - **Interacts:** a more general relation for usage or dependency (e.g. a function using a type, or a module using another module's API).

This **interface graph** forms a **minimal map of the codebase's structure** – essentially capturing the *wiring* of the system without full implementations. It enables queries like "who calls this function?", "which types implement this trait/interface?", or "where is this data type used?". Having such an explicit graph is crucial because **code is not just text; it's structured knowledge** – more like an *interconnected web of symbols* than a linear document [5]. Many modern AI code tools have realized that *"code's meaning isn't in how it reads, but in how it connects"* [6] [7]. In other words, the relationships between functions, types, and modules define the system's behavior.

**Benefits:** A knowledge graph of the code provides comprehensive context that an LLM can leverage. Instead of relying on fuzzy text search, the LLM (or a developer) can traverse the exact dependency graph. This addresses a known challenge: *LLMs need complete information about dependencies and call relationships across the codebase to reason accurately* [8]. Static retrieval (RAG) based on text similarity often fails for code, because a small code change can have ripple effects far away with no obvious textual similarity [7]. By contrast, a graph of explicit links lets us see those *"invisible relationships"* directly [7]. For example, using the graph we can quickly find the **blast radius** of a change: if we alter a function's signature, the graph reveals all callers (who_calls), all interfaces it touches, etc., preventing surprises. This approach is akin to what code analysis tools and language servers do – they build a symbol graph so you can *navigate code structure with 100% precision (jump to definitions, find references, list implementors)* [9] [10]. In fact, treating the codebase as a graph is increasingly seen as the right abstraction: *"codebases are graphs, not documents"*, and understanding code means having a **map of those connections** [11] [12].

**Implementation:** We can compute the graph from the interface spec (Idea 1) or from code stubs. Each function signature (node) would carry a unique SigHash ID (for example, a hash of the name and parameter types) to make it easy to reference. Then we populate edges: - A **calls** edge for each function invocation (from caller to callee) identified in the spec. - An **implements** edge for each type fulfilling an interface/trait contract. - An **interacts** edge for other uses (for example, function X uses Type Y as a parameter or return, or module-level relationships).

Such a graph can be stored compactly (as adjacency lists or even in a single JSONL where each line describes a relation triplet). Notably, building graphs over code isn't new – static analysis frameworks routinely create call graphs, ASTs, etc., for compilers or vulnerability analyzers [13]. Our twist is focusing on an *abstract interface graph* that's minimal but sufficient for reasoning. This graph becomes the backbone for subsequent tools: we can answer *who_calls*, *who_implements*, *who_interacts* queries instantly by traversing it. For example, using the graph, an LLM or tool can find "all functions that call X" or "all modules that would be impacted if we change trait Y" (blast radius analysis). This ensures **full coverage**: the spec plus graph help verify that every requirement and connection is accounted for, and highlight any orphaned components or cyclic dependencies (potential bugs) before coding begins.

## 3. Graph Query Tool (Rust-based CLI for LLMs)

**Idea:** Provide a dedicated **graph query operator** – essentially a command-line tool, likely written in Rust for efficiency – that an LLM agent can invoke to analyze the interface graph. This tool would accept queries (possibly in a simple query language or structured format) and return answers or subgraph data. The LLM, during code generation or analysis, could call this CLI to ask questions like "find all callers of function X" or "list the traits implemented by type Y", etc. By using Rust, the tool can efficiently handle large graphs in memory and perform complex queries quickly, and also integrate with Rust's powerful parsing libraries for code if needed. The CLI could output results as JSON or a concise text summary that the LLM can parse.

**Purpose:** The graph operator serves as the LLM's **eyes into the code structure**. It's analogous to giving the LLM an API similar to an IDE's "Go to Definition" or "Find References" capability. In fact, this mirrors the approach of augmenting LLMs with Language Server Protocol (LSP) features – LSPs build a graph of symbols and allow queries on definitions, references, implementations [9]. By exposing a custom tool, we enable *deterministic traversal of code relationships*, which is far more precise than embedding-based search [14]. Chinmay Bharti (CodeAnt AI) emphasizes that for code review by AI, *"we need to traverse exact relationships, not approximate them"* [14]. The CLI would ensure the LLM can retrieve those exact relationships on demand.

**Use Cases:** During development, the LLM might generate code for a function and want to ensure it doesn't break any contracts – it could call the graph tool to fetch all places that function is used (to then load their context or tests). Or if asked a question about how data flows, the LLM could query the path in the graph. The tool can also simulate **behavior** by walking the graph (e.g., follow calls from an entry-point function to see what it ultimately reaches, which might help estimate performance or identify missing error handling along a call chain). In essence, the graph operator acts as a *local knowledge base* about the design that the LLM can tap into whenever its context isn't enough.

**Integration:** We can integrate this Rust CLI via a standard protocol. For example, the emerging **Model Context Protocol (MCP)** is designed to let AI tools communicate with context providers in a standardized way [15]. An implementation of a context server in Rust already exists that uses STDIO for communication and can be discovered by clients like IDEs or chat interfaces [16] [17]. We could implement our graph query tool as an MCP server or a similar service, so that any LLM interface (VSCode extension, chat app, etc.) can call it. The tool would load the interface graph (from file or a database) and respond to queries. By using Rust, we also ensure the tool can be open-sourced with minimal dependencies and be performant for real-time usage. Overall, this gives the LLM *superpowers* to reason about the codebase like a developer armed with an IDE, making the development process smarter and safer.

## 4. Context Retrieval via SQLite and JSONL

**Idea:** Store the interface knowledge (from Idea 1 and 2) in a **local SQLite database** (or a set of JSONL files indexed by SQLite) to enable flexible querying and retrieval of context. SQLite is lightweight, file-based, and now has good JSON support – meaning we can store JSON documents (like spec entries or graph edges) and query them with SQL. The LLM (or the aforementioned Rust tool) can use SQL queries to fetch a **bounded context** around a specific node of interest. For instance, when generating code for a particular function, we might query SQLite for that function's signature and all directly related entities (calls in/out, the trait it implements, the data types it touches). This would return a *terminal slice* of the graph – a localized subgraph that is highly relevant to the task at hand – which can then be supplied to the LLM as additional context or hints for code generation.

**Benefits:** Using SQLite for this purpose has multiple advantages: - *Embedded and Fast:* SQLite can run locally without a server, and efficiently query even thousands of relationships. - *Structured Queries:* By formulating SQL (possibly generated by the LLM itself with a predefined schema), we can precisely retrieve the needed info (no hallucination – the query results are exact). This is similar to how some advanced code assistants operate: they maintain a structured index or graph of the codebase and let the LLM query it for relevant pieces [18] [19]. - *JSONL + SQL:* We could store each interface or edge as a JSON row and use SQLite's JSON functions to filter by fields (e.g., find all edges where `source = "FunctionX"`). Alternatively, maintain separate tables for Functions, Types, Traits, and Edges for more relational queries. The simplicity of SQLite also makes it easy to update as the spec evolves or to integrate into build pipelines.

**Precedents:** This approach aligns with recent developments in AI tooling. In fact, an open-source **MCP Context Server** (for AI code generation) uses embedded SQLite to store project context and serve it to AI agents [16]. It allows creating a project database and querying it for relevant components or features. By storing our interface graph in a SQLite DB, we set up a mini knowledge base that can answer detailed questions. For example, an LLM could ask (via a tool or by generating SQL): "Which functions in module X call database API and what are their expected latencies?" and get back the list of functions and perhaps a stored annotation of their latency budgets. This is far more scalable than packing the entire codebase into the prompt. It's a form of **Retrieval-Augmented Generation (RAG)** but using *structured retrieval* instead of just semantic search. As Simon Willison notes, long-context support is improved by breaking code into fragments and retrieving relevant ones on demand [20] [21] – a knowledge graph + SQLite lets us do this slicing by following logical connections rather than arbitrary text chunks.

**Usage in Code Generation:** When the LLM is ready to implement a particular interface stub, a tool could automatically run a SQLite query to gather *just enough* context (related type definitions, any policy annotations, calls it should make, etc.). This context can be prepended to the LLM's prompt, guiding the code generation. The result is **focused, context-aware code synthesis** – the LLM sees the local neighborhood of the function in the design graph, ensuring consistency (for example, using the right data models and abiding by stated invariants). As our design scales, SQLite provides a robust backbone to manage this data-driven approach to code generation.

## 5. Cross-Stack Micro Graph for User Flows (End-to-End Specification)

**Idea:** Extend the interface graph concept beyond just back-end code relationships to capture **cross-stack interactions** – essentially a *holistic micro-graph* covering front-end, back-end, and any external services. In a

modern application, a user action might travel through multiple layers (UI → API → service → database, etc.). We want to map these **UX flows before coding**, so that the end-to-end behavior is specified and any integration points are clear. We can introduce additional node and edge types for this cross-stack map: - **Nodes:** Could include UI components/pages, API endpoints (routes), microservice handlers, database or external service endpoints. - **Edges:** For example: - *handles:* A UI element or client action that is handled by a certain front-end function. - *fetches:* A front-end call to a back-end API (e.g., an edge from a web page or JS function to an HTTP route). - *routes_to:* Mapping from an API route to the server-side handler function that implements it. - *mounts:* The relationship of attaching a handler to a route or a middleware mount point (useful in frameworks). - *triggers / publishes:* If using events or message queues, edges for events emitted and caught.

By mapping these out, we essentially create a **graph of graphs**: tying the user interface to the underlying logic graph. For each critical user journey (e.g., "Sign Up Flow", "Checkout Process"), we can chart the exact path and alternatives (including "detours" like error paths or third-party calls). This ensures that **UX flows are specified and reviewed up front**, preventing surprises later.

**Importance:** Capturing cross-stack flows is vital for building resilient systems. Often, bugs or performance issues appear not in isolated functions but in the interactions between systems. For example, a slight delay in a third-party service or an unexpected retry loop can cascade failures. As one SRE put it, if you don't map the journey exactly – *"detours and all"* – you might miss how a small change causes chaos downstream [22]. By drawing a micro-level architecture graph, we incorporate those detours: maybe a mobile app retries a request twice, or an optional OAuth flow adds an extra round-trip – these can be represented as branches or additional edges in the scenario.

**Visualization & Analysis:** Once we have this cross-stack micro-graph, we can analyze **end-to-end performance budgets**. Each edge could carry a latency or resource cost (for instance, mark external API calls with an expected 95th percentile time). Then the sum along a path yields the expected p99 latency of the whole user flow. If it exceeds our budget, we've caught a problem *before any code is written*. Similarly, we can ensure **consistency and idempotency** across systems: for example, mark that a certain API call is idempotent and verify via the graph that it might be retried safely (no multiple inserts, etc.), or that a UI will not double-submit actions. The cross-stack graph also assists in **coverage analysis**: each user scenario in the spec should correspond to a path in this graph – if a path is not covered, maybe a user interaction is missing in our design.

**Tooling:** We can incorporate these cross-layer edges into our JSONL spec as well. For instance, an entry might say: `{"edge": "fetches", "from": "UI.Screen.Payment", "to": "API.Endpoint.ChargeCard"}`. We could then auto-generate sequence diagrams or flow charts. Indeed, tools like Zencoder's AI can produce **user journey flow diagrams and service interaction maps** from a repository understanding [23]. By having the spec, we could generate a Mermaid sequence diagram for each user flow to visually inspect it (Idea 6). This cross-stack graph also encourages thinking in terms of **services and contracts** – very useful if the project is microservice-based or has multiple teams. Everyone can see how data and commands propagate through the whole system, in one connected blueprint.

## 6. Visualizing Relationships with Mermaid Diagrams

**Idea:** Leverage **Mermaid diagrams** (a text-based diagramming tool supported in Markdown) to visualize the relationships defined by our interface signatures and graphs. After constructing the interface graph

(and cross-stack extensions), we can auto-generate diagrams for easier human understanding. This might include: - **Class/Interface Diagrams:** showing types, traits, and implementation relationships. - **Call Graph Diagrams:** a flowchart or graph of function calls for a given module or the entire system (likely filtered to high-level calls to avoid over-crowding). - **Component Diagrams:** for cross-stack, showing how front-end, back-end, and external components interact. - **Sequence Diagrams:** for specific scenarios or user flows (each scenario from Idea 5 can be turned into a sequence chart of interactions).

Using Mermaid syntax, we can programmatically produce these diagrams from the spec. The diagrams will help us and stakeholders spot potential issues (e.g., circular dependencies, or an oddly connected component) and verify understanding.

**Advantages:** Visualizing the architecture is invaluable for identifying bugs *before* they happen. By seeing a diagram of all relationships, architects might notice, for example, that a certain module has too many responsibilities (many arrows pointing to it), or that two subsystems unexpectedly depend on each other (introducing a cycle). Mermaid diagrams are easy to generate and include in docs, meaning our design can be documented in a living manner. Because the diagrams come *from the spec/graph*, they stay up-to-date. This addresses a common documentation problem where architecture diagrams drift from reality – here, if the code spec changes, regenerate the Mermaid diagrams to get the updated picture.

**Feasibility:** Tools have already demonstrated the power of AI-driven diagram generation. For instance, Zencoder's coding assistant can analyze a codebase and output a Mermaid chart reflecting the actual architecture [24]. The key is that it *"understands how everything fits together"* from repository analysis and produces a diagram that matches the real relationships [24]. We would be doing the same, but using our interface graph as the source of truth. Mermaid supports many diagram types (flowcharts, sequence diagrams, class diagrams, entity relationships, etc.), so we can choose whichever best represents a given aspect. For a high-level overview, we might generate a **module dependency graph**; for detail, a **class diagram** of a particular module's internals.

When embedding these diagrams in our Markdown docs or wikis, they become powerful communication tools. A developer joining the project can glance at the Mermaid diagrams and quickly grasp the structure (rather than reading a 100-page spec). And since generating them can be automated via a script, this could be integrated into CI: e.g., whenever the spec changes, regenerate diagrams. This fosters transparency and shared understanding, ultimately reducing bugs caused by miscommunication. As an example of the capability, one could even prompt an AI agent with *"Generate a Mermaid diagram of the entire repository's architecture"* and get a comprehensive chart [25] – we aim to have that level of overview from our spec.

## 7. Obsidian-Style Graph View for Code Connectivity

**Idea:** Provide an interactive **graph visualization** of the interface relationships, similar to Obsidian's graph view for notes. In Obsidian (a knowledge base tool), you can see all notes as nodes and their links as connections in a force-directed graph [26]. We can mimic this for our code interfaces: each function, type, or module could be a "node" (perhaps represented as a small markdown note automatically generated), and links (calls, implements, etc.) become edges. Using a tool or plugin, we could navigate this graph visually – clicking on a node to see its connections, filtering by type of relationship, etc.

**Benefits:** This visual graph serves as a **living architecture map**. Early in design, it helps us validate the structure (for example, visually cluster nodes by module to see if any unexpected cross-module links

appear). Later, during coding or code review, a developer could use it to quickly answer questions like "what modules will this change affect?" by literally seeing the network of edges from the relevant node. It's a more free-form complement to the formal Mermaid diagrams – Mermaid diagrams might be curated for documentation, whereas an Obsidian-style graph is an interactive exploration tool.

Obsidian itself might be repurposed: one could generate an Obsidian vault where each interface or component is a note containing a short description and links (in Obsidian Markdown, linking notes via `[[NoteName]]`). For example, the note for a function **Foo** might contain `[[Bar]]` in text if Foo calls Bar. Obsidian's graph view would then automatically show Foo connected to Bar. This approach piggybacks on an existing tool to give us a rich visualization without coding a new UI. Alternatively, other open-source graph visualization libraries (like **D3.js**, **Graphviz**, or **Gephi**) could be fed our graph data for more custom visual analysis.

**Use Cases:** Imagine being able to filter the graph to show only "implements" edges to see the interface-implementation hierarchy at a glance, or filter to only show a particular user flow's nodes (highlighting the path a request takes through the system). If an anomaly is present – say a utility module from deep in the back-end is somehow being referenced by a UI component (violating layering) – it would stick out on the graph, prompting a design correction. The visual graph also makes team discussions easier: instead of guessing relationships, the team can literally point to them. It fosters a *shared mental model* of the project's structure.

**Feasibility:** This idea is quite feasible given that Obsidian's graph view is readily available and highly configurable. We'd mostly need to script the generation of markdown notes from our spec. Each note could list the interfaces it connects to (incoming/outgoing edges). Then Obsidian (or a similar markdown knowledge base) does the rest. This can be part of our open-source toolkit – we publish the spec and also an **Obsidian vault** as part of documentation. The community could open it and instantly explore the project's design graph. Overall, an Obsidian-style graph offers an intuitive, zoomable way to understand complex relationships, complementing the static Mermaid diagrams.

## Open Source Implementation and Next Steps

All the above ideas lend themselves well to an **open-source project**, combining components that are themselves open or standard. Here's how we can proceed and leverage open source:

- **Leverage Existing Parsers/Analyzers:** Instead of writing everything from scratch, we can utilize open-source parsing libraries (e.g., Tree-sitter for multi-language ASTs, or language-specific analyzers like Rust's `rust-analyzer` for building the symbol graph). Language Server Protocol (LSP) implementations are open source and already *"build a complete graph of your codebase (symbols, references, types)"* [10] . We can tap into those to extract the initial function/type graph systematically. This speeds up development and ensures accuracy.
- **Graph Storage and Query:** For the graph database, **Neo4j** or other graph DBs could be used, but to keep things simple and lightweight, we can stick with **SQLite** (which is public domain/open source). The choice of JSONL+SQLite means anyone can run the tooling locally without complex setup. It's noteworthy that projects like the MCP context server we discussed are MIT-licensed and use SQLite for similar purposes [16] , so we have reference implementations to consult. We can open source our schema for the context database so others can contribute improvements (for example, adding new query types or optimizations).

- **Rust CLI Tool:** We plan to implement the graph query CLI in Rust and release it on GitHub under a permissive license. Rust has a robust ecosystem for command-line tools and assured performance. Being open-source will allow others to extend the query language (perhaps adding support for more complex queries or even pathfinding algorithms in the call graph). It can also integrate with other AI tools; by following the Model Context Protocol spec (which is open) [17], our tool could be discovered by any AI assistant that implements MCP, fostering interoperability.
- **Diagram Generation:** The scripts to generate Mermaid diagrams or Obsidian vault content from the spec will be open source as well. This invites collaboration – e.g., the community might contribute improvements to the visualization (maybe a more sophisticated grouping in the Mermaid diagrams, or additional diagram types). Mermaid itself is open source, and any generated diagrams can be viewed without proprietary software.
- **Community and Contributions:** By open-sourcing the entire approach (spec format, graph tool, context DB, visualization scripts), we encourage a community to form around **LLM-driven development tooling**. Developers could contribute support for new languages (extending the parser for the spec to generate code in Python, Go, etc.), or improve analysis (for example, adding a static analyzer that reads the spec and warns if a performance budget is likely exceeded by a given call chain). There is already growing interest in repository-level code understanding [27], and our project could fit into that landscape as a pragmatic toolset.

In conclusion, this **interface-stub and graph-driven approach** offers a pathway to build complex systems with high assurance and LLM assistance. By compressing the design into a clear, analyzable form, we gain the ability to examine the system's properties (coverage, invariants, performance) with both human insight and machine support *before* implementation. The ideas of building knowledge graphs over code and integrating them with LLM workflows are being validated by current research and tools [8] [28]. Our plan is to bring these techniques together into a cohesive, open-source solution. This would empower developers to confidently use LLMs in coding – not as a blind code generator, but as part of a **smart, guided development process** where an interface spec and its knowledge graph keep everything on track. By "designing in the open" and visualizing relationships, we make our greenfield project far more robust from day one, and we invite the open-source community to join in refining this new way of building software.

**Sources:**


1. Harper Reed – *"My LLM codegen workflow"* (Feb 2025) – on brainstorming specs and planning before code [1] [2] [4].
2. Reddit (ChatGPTCoding) – user discussion of multi-phase IDL-to-code generation, illustrating interface-first approach [3].
3. Chinmay Bharti – *"How Call Graphs Gave Our LLM Superhuman Code Review Context"* (Jul 2025) – emphasizes precise call graph mapping for full context [8] [14].
4. CodeAnt AI Blog – *"LLM-Powered Code Reviews Go Beyond RAG"* (May 2025) – discusses code as structured graph of relationships, and using LSP-like knowledge for AI [7] [5] [9] [10].
5. Zimin Chen – *"Building Knowledge Graph over a Codebase for LLM"* (Jun 2024) – shows how a code knowledge graph enables complex queries and better context for LLMs [21] [18] [28].
6. Zencoder AI Docs – *"Generate codebase architecture diagrams with Mermaid"* (2023) – demonstrates automated diagram generation from repo understanding [24] [23].
7. Chaos Engineering Field Guide – *"Identify and Map Critical User Journeys (CUJs)"* (Sep 2025) – highlights the need to map full user flows with detours to avoid blind spots [22].

8. Model Context Protocol (MCP) – *Rust Context Server for AI Code Generation* (Sep 2025) – uses embedded SQLite to serve project context to LLMs, an open-source example of structured context retrieval [16] [15].
9. Obsidian Help – *Graph View* – explains Obsidian's graph visualization of relationships between notes [26].

---


[1] [2] [4] My LLM codegen workflow atm | Harper Reed's Blog
https://harper.blog/2025/02/16/my-llm-codegen-workflow-atm/

[3] Complex code generation in LLMs : r/ChatGPTCoding
https://www.reddit.com/r/ChatGPTCoding/comments/1d8b1hw/complex_code_generation_in_llms/

[5] [6] [7] [9] [10] [11] [12] Why LLM-Powered Code Reviews Go Beyond RAG (And Why You Should Too) - CodeAnt AI
https://www.codeant.ai/blogs/llm-code-reviews-beyond-rag

[8] [14] How Call Graphs Gave Our LLM Superhuman Code Review Context | by Chinmay Bharti | Jul, 2025 | Level Up Coding
https://levelup.gitconnected.com/how-call-graphs-gave-our-llm-superhuman-code-review-context-df735a1d8d88?gi=48250ea4eaad

[13] [18] [19] [20] [21] [27] [28] Building Knowledge Graph over a Codebase for LLM | by Zimin Chen | Medium
https://medium.com/@ziche94/building-knowledge-graph-over-a-codebase-for-llm-245686917f96

[15] [16] [17] MCP Context Server for AI Code Gener... · LobeHub
https://lobehub.com/mcp/hrirkslab-context-server-rs

[22] CERA Best Practices, Part-2: Identify and Map Critical User Journeys (CUJs) | by ABHISHEK DATTA | Chaos Engineering Field Guide — Experiments, Learnings, and Best Practices. | Sep, 2025 | Medium
https://medium.com/chaos-engineering-field-guide-experiments/cera-best-practices-part-2-identify-and-map-critical-user-journeys-cujs-2b41d321e855

[23] [24] [25] Generate codebase architecture diagrams with Mermaid - Zencoder Docs
https://docs.zencoder.ai/user-guides/tutorials/generate-codebase-diagrams

[26] Graph view - Obsidian Help
https://help.obsidian.md/plugins/graph