# CodeBase-Xray-Prompt

Analyze the entire provided codebase (approximately 50,000+ lines spanning multiple files and folders) and output a \*\*compact, near-lossless JSON representation\*\* of the system's architecture, all code entities, and their interconnections. \*\*Follow the instructions below step-by-step with absolute thoroughness and specificity.\*\* Assume no prior context beyond the given code, and explicitly perform each step to ensure nothing is overlooked.

## 1. Absolute Granularity & Specificity

- \*\*Identify \*every\* relevant element\*\* in the codebase. Do not skip any file or code construct. Treat each file independently at first, deriving all information purely from its content.

- \*\*Be extremely specific\*\* in what you report: capture names, definitions, and details exactly as they appear. The goal is a near-lossless capture of the codebase's structure.

## 2. Complete Component Inventory (per File)

For \*\*each file\*\* in the codebase, compile a comprehensive list of all code components defined in that file. This includes (but is not limited to):

- \*\*Functions\*\* (free-standing or static functions)

- \*\*Methods\*\* (functions defined as part of classes or structs)

- \*\*Classes\*\* (including any nested or inner classes)

- \*\*Structs\*\* (data structures, if applicable in the language)

- \*\*Interfaces\*\* (interface or protocol definitions)

- \*\*Variables\*\* (global variables, module-level variables, class-level attributes, instance attributes, and significant local variables)

- \*\*Constants\*\* (constant values, enums, or read-only variables)

- \*\*Imports\*\* (import/include statements with their origins. Each import can be listed as an entity of kind "import", including the module or symbol name and source module/package)

- \*\*Exports\*\* (export statements, each as an entity of kind "export" with the symbol being exported)

- \*\*Decorators/Annotations\*\* (function or class decorators, annotations above definitions)

- \*\*API Routes\*\* (web or API endpoints. Each route can be an entity of kind "route" with the route path or identifier as its name)

- \*\*Configuration References\*\* (usage of configuration settings or environment variables. Each distinct config key used can be an entity of kind "config\_ref")

For each identified component, \*\*capture all of the following details\*\*:

- \*name\*: the identifier/name of the entity.

- \*kind\*: the type of entity (e.g. `"file"`, `"package"`, `"module"`, `"class"`, `"struct"`, `"interface"`, `"function"`, `"method"`, `"variable"`, `"constant"`, `"import"`, `"export"`, `"decorator"`, `"route"`, `"config\_ref"`).

- \*scope\*: where this entity is defined or accessible. Use `"global"` for truly global items, `"module"` for file-level (top-level) items within a file/module, `"class"` for class-level (static or class variables/methods inside a class), `"instance"` for instance-level (non-static class members or object instances), or `"local"` for local scope (variables inside a function).

- \*signature\*: the definition details. For functions/methods, include parameters and return type or description (e.g. `functionName(param1, param2) -> ReturnType`). For classes/interfaces, you might list base classes or implemented interfaces. For variables/constants, include their type or value if evident (e.g. `PI: Number = 3.14`). Keep it concise but informative.

- \*visibility\*: the access level (if the language uses it), such as `"public"`, `"private"`, `"protected"`, or similar. If not explicitly provided by the language, infer based on context (e.g. assume module-level functions are public if exported, otherwise internal). If not applicable, you can omit or use a default like `"public"`.

- \*line\_start\* and \*line\_end\*: the line numbers in the file where this entity’s definition begins and ends.

Ensure this inventory covers \*\*every file and every entity\*\* in the codebase.

## 3. Deep Interconnection Mapping

Next, \*\*map all relationships and interactions\*\* between the entities across the entire codebase. For each relationship where one entity references or affects another, create a relationship entry. The relationships should precisely capture:

- \*\*Function/Method Calls\*\*: Identify every time a function or method (`from`) calls another function or method (`to`). Mark these with `type: "calls"`.

- \*\*Inheritance\*\*: If a class extends/inherits from another class, use `type: "inherits"` (from subclass to superclass). If a class implements an interface or protocol, use `type: "implements"` (from the class to the interface).

- \*\*Instantiation\*\*: When a function or method creates a new instance of a class (i.e. calls a constructor or uses `new`), use `type: "instantiates"` (from the function/method to the class being instantiated).

- \*\*Imports/Usage\*\*: If a file or module imports a symbol from another, represent it as `type: "imports\_symbol"` (from the importer entity or file to the imported entity’s definition). Additionally, if an imported symbol is later used in code (e.g. a function uses a function from another file that was imported), denote that with `type: "uses\_imported\_symbol"` (from the place of use to the imported symbol’s entity).

- \*\*Variable Usage\*\*: When a variable defined in one scope is read or accessed in another, use `type: "uses\_var"` (from the usage location to the variable’s entity). If a variable is being written or modified, use `type: "modifies\_var"`.

- \*\*Data Flow / Returns\*\*: If a function returns data that is consumed by another component, denote it as `type: "returns\_data\_to"` (from the function providing data to the consumer). For example, if function A’s return value is passed into function B, or if a function returns a result that an API route sends to the client, capture that flow.

- \*\*Configuration Usage\*\*: If code references a configuration setting or environment variable, use `type: "references\_config"` (from the code entity to the config reference entity).

- \*\*API Route Handling\*\*: If an API route is associated with a handler function, use `type: "defines\_route\_for"` (from the route entity to the function that handles that route).

- \*\*Decorators\*\*: If a function or class is decorated by another function (or annotation), use `type: "decorated\_by"` (from the main function/class entity to the decorator function’s entity).

Each relationship entry should include:

- \*from\_id\*: the unique id of the source entity (the one that references or calls or uses another).

- \*to\_id\*: the unique id of the target entity (the one being called, used, inherited from, etc.).

- \*type\*: one of the above relationship types (`"calls"`, `"inherits"`, `"implements"`, `"instantiates"`, `"imports\_symbol"`, `"uses\_imported\_symbol"`, `"uses\_var"`, `"modifies\_var"`, `"returns\_data\_to"`, `"references\_config"`, `"defines\_route\_for"`, `"decorated\_by"`).

- \*line\_number\*: the line number in the source file where this relationship occurs (e.g. the line of code where the function call or import is made).

Map \*\*every occurrence\*\* of these relationships in the codebase to ensure the JSON details how all parts of the code connect and interact.

## 4. Recursive Chunking and Synthesis for Large Contexts

Because the codebase is large, use a \*\*divide-and-conquer approach\*\* to manage the analysis:

\*\*(a) Chunking:\*\* Break down the input codebase into manageable chunks. For example, process one file at a time or one directory at a time, ensuring each chunk fits within the model’s context window. Do not split logical units across chunks (e.g. keep a complete function or class within the same chunk).

\*\*(b) Chunk Analysis:\*\* Analyze each chunk independently to extract a structured summary of its entities and relationships (as defined in steps 2 and 3). Treat each chunk in isolation initially, producing partial JSON data for that chunk.

\*\*(c) Hierarchical Aggregation:\*\* After processing all chunks, merge the results. First combine data for any files that were split across chunks. Then aggregate at a higher level: integrate all file-level summaries into a complete project summary. Construct a hierarchical \*\*file\_structure\*\* (directory tree) from the file and folder names, and consolidate the lists of entities and relationships from all chunks.

\*\*(d) Global Synthesis & Cross-Linking:\*\* Now, examine the aggregated data and connect the dots globally. Deduplicate entities that are identical (ensure each unique function/class/variable appears only once with a single id). Resolve cross-file references: if an entity in one file references another in a different file (for example, calls a function defined elsewhere), make sure there is a relationship linking their ids. Merge any relationships that span chunks. The result should be a coherent global map of all entities and their interconnections across the entire codebase.

\*\*(e) Iteration (Optional):\*\* If inconsistencies or missing links are found during global synthesis, iterate to refine. Re-check earlier chunk outputs with the new global context in mind. For instance, if you discover an import in one chunk corresponds to a function defined in another, ensure that function’s entity exists and add the appropriate relationship. Only re-analyze chunks as needed to fill gaps or resolve ambiguities, avoiding redundant re-processing of unchanged content. Continue iterating until the global model is consistent and complete.

## 5. Advanced Reasoning Techniques

Employ advanced reasoning to ensure the analysis is correct and comprehensive:

- \*\*Tree-of-Thought (ToT) Reasoning:\*\* During global synthesis, systematically explore multiple reasoning paths for how components might relate. Consider different possible interpretations for ambiguous cases (for example, a function name that appears in two modules—determine which one is being referenced by considering both possibilities). By exploring these branches of thought, you can discover hidden connections or confirm the correct architecture. After exploring, converge on the most coherent and evidence-supported interpretation of the relationships.

- \*\*Self-Consistency Checks:\*\* For complex sections of the code or uncertain relationships, perform internal self-consistency checks. Imagine analyzing the same part of the code multiple times (e.g. in different orders or with slight variations in assumptions) and observe the conclusions. If all these hypothetical analyses agree on a relationship (e.g. they all conclude function X calls function Y), you can be confident in that result. If there are discrepancies, investigate why and choose the interpretation that is most consistent with the actual code content. This approach of cross-verifying results will reduce errors and improve the reliability of the final output.

## 6. Robustness and Error Handling

Ensure the process and output are resilient and correct:

- \*\*Validate JSON Schema:\*\* After constructing the final JSON, verify that it strictly conforms to the required schema (see section 7). All keys should be present with the correct data types. The JSON should be well-formed (proper brackets and commas) and pass a JSON parser.

- \*\*Auto-Repair if Needed:\*\* If any structural issues or schema deviations are detected in the JSON (e.g. a missing field, a null where an array is expected, or a parse error), automatically fix them before finalizing. The goal is to output a clean JSON that requires no manual corrections.

- \*\*Truncation Handling:\*\* If the output is extremely large, ensure it isn’t cut off mid-structure. If you must truncate, do so gracefully: for example, close any open JSON structures and perhaps add a note or flag indicating that the output was abbreviated. However, the preference is to produce a \*compact\* yet information-rich JSON, so truncation should ideally be avoided by summarizing repetitious structures.

- \*\*Avoid Redundancy:\*\* Do not repeat analysis unnecessarily. If you have already analyzed a chunk or identified certain entities/relationships, reuse that information. This is especially important if iterative refinement is used—skip re-analyzing code that hasn’t changed. This will help keep the output concise and prevent inconsistent duplicate entries.

## 7. Required Output Format

Finally, present the results in a \*\*single JSON object\*\* that captures the entire codebase analysis. The JSON \*\*must strictly follow\*\* this schema structure (with exact keys and nesting as specified):

{

"schema\_version": "1.1",

"analysis\_metadata": {

"language": "[Inferred or Provided Language]",

"total\_lines\_analyzed": "[Number]",

"analysis\_timestamp": "[ISO 8601 Timestamp]"

},

"file\_structure": {

"path/to/dir": { "type": "directory", "children": [...] },

"path/to/file.ext": { "type": "file" }

},

"entities": [

{

"id": "<unique\_entity\_id>",

"path": "<relative\_file\_path>",

"name": "<entity\_name>",

"kind": "<file|package|module|class|struct|interface|function|method|variable|constant|import|export|decorator|route|config\_ref>",

"scope": "<global|module|class|instance|local>",

"signature": "<params\_and\_return>",

"line\_start": "[Number]",

"line\_end": "[Number]"

}

// ... more entities ...

],

"relationships": [

{

"from\_id": "<unique\_entity\_id\_source>",

"to\_id": "<unique\_entity\_id\_target>",

"type": "<calls|inherits|implements|instantiates|imports\_symbol|uses\_imported\_symbol|uses\_var|modifies\_var|returns\_data\_to|references\_config|defines\_route\_for|decorated\_by>",

"line\_number": "[Number]"

}

// ... more relationships ...

]

}

- \*\*schema\_version\*\*: use `"1.1"` exactly.

- \*\*analysis\_metadata\*\*: provide the programming `"language"` (inferred from the code, or provided explicitly), `"total\_lines\_analyzed"` (the sum of lines of all files processed), and an `"analysis\_timestamp"` (the current date/time in ISO 8601 format, e.g. `"2025-05-04T18:07:16Z"`). You may include additional metadata fields if useful (e.g. number of files), but these three are required.

- \*\*file\_structure\*\*: a hierarchical mapping of the project’s files and directories. Each key is a path (relative to the project root). For each directory, set `"type": "directory"` and include a `"children"` list of its entries (filenames or subdirectory paths). For each file, set `"type": "file"`. This provides an overview of the codebase structure.

- \*\*entities\*\*: an array of entity objects, each describing one code entity discovered (as detailed in step 2). Every function, class, variable, import, etc. should have an entry. Ensure each entity has a unique `"id"` (for example, combine the file path and the entity name, and if necessary a qualifier like a class name to disambiguate). The `"path"` is the file where the entity is defined. The `"name"`, `"kind"`, `"scope"`, `"signature"`, and line numbers should be filled out as described.

- \*\*relationships\*\*: an array of relationship objects, each representing an interaction between two entities (as detailed in step 3). Use the `"id"` values of the entities for `"from\_id"` and `"to\_id"` to refer to them. `"type"` must be one of the specified relationship types. The `"line\_number"` is where the interaction is found in the source.

\*\*The output should be a single valid JSON object\*\* following this format. Do not include any narrative text outside of the JSON structure (except the optional summary in section 9). The JSON should stand on its own for programmatic consumption.

## 8. Concrete Language-Agnostic Example

To illustrate the expected output format, consider a simple example in a generic programming language:

\*\*Input (example code):\*\*

// File: src/math/utils.[ext]

export function add(a, b) {

return a + b;

}

\*(This represents a file `src/math/utils.[ext]` containing one exported function `add`.)\*

\*\*Expected JSON fragment (for the above input):\*\*

{

"entities": [

{

"id": "src/math/utils.[ext]:add",

"path": "src/math/utils.[ext]",

"name": "add",

"kind": "function",

"scope": "module",

"signature": "(a, b) -> return a + b",

"line\_start": 1,

"line\_end": 3

}

],

"relationships": []

}

In this fragment, we see one entity for the `add` function with its details. There are no relationships because `add` does not call or use any other entity in this snippet. \*\*This example is language-agnostic\*\* – the prompt should work similarly for any language, capturing analogous details (e.g. functions, classes, etc. in that language).

## 9. Executive Summary (Optional)

After producing the JSON output, you may append a brief \*\*Executive Summary\*\* in plain English, summarizing the codebase. This should be a high-level overview (at most ~300 tokens) describing the overall architecture and important components or interactions. If included, prepend this summary with a clear marker, for example:

Executive Summary

<Your 1–2 paragraph overview here>

This section is optional and should only be added if an overview is needed or requested. It comes \*\*after\*\* the closing brace of the JSON. Ensure that adding the summary does not break the JSON format (the JSON should remain valid and complete on its own).

\*\*Final Output Requirements:\*\* Generate the final output strictly as specified:

- Output the \*\*JSON object only\*\*, following the schema in section 7, representing the full codebase analysis.

- Optionally include the executive summary section after the JSON (as unstructured text, not part of the JSON).

- Do \*\*not\*\* include any extra commentary, explanation, or formatting outside of these. The response should be the JSON (and summary if used) and nothing else.