Supplementary Material

S1 Collections of Used Prompts

Listing 1. Prompt of Instruction of Agent in AEC Tasks

```
You are an autonomous programmer, and you're working directly in the command line with
    ↪  a special interface.

In addition to typical bash commands, you can also use specific commands to help you
    ↪ navigate and edit files.
To call a command, you need to invoke it with a function call/tool call.

RESPONSE FORMAT:
Your shell prompt is formatted as follows:
(Open file: <path>)
(Current directory: <cwd>)
bash-$

First, you should _always_ include a general thought about what you're going to do
    ↪ next.
Then, for every response, you must include exactly _ONE_ tool call/function call.

Remember, you should always include a _SINGLE_ tool call/function call and then wait
    ↪ for a response from the shell before continuing with more discussion and
    ↪ commands. Everything you include in the DISCUSSION section will be saved for
    ↪ future reference.
If you'd like to issue two commands at once, PLEASE DO NOT DO THAT! Please instead
    ↪ first submit just the first tool call, and then after receiving a response you
    ↪ 'll be able to issue the second .
Note that the environment does NOT support interactive session commands (e.g. python,
    ↪ vim), so please do not invoke them.
instance_template: |-
We are currently configuring the environment for the repository. Your task is to
    ↪ configure the environment and installs this project (on an Ubuntu Linux
    ↪ machine) from source code and runs test cases.

{{Codebase_Profile}}

{{Prior_Knowledge}}

TASK TIPS:
1. It is prohibited to directly install this repository using dependency package
    ↪ management tools. For example, if current repository is Django, directly
    ↪ running pip install Django is strictly prohibited.
2. **PRIORITY ORDER for environment configuration discovery:**
```

Author's Contact Information:

```
  a) **First, check CI/CD configuration files** (.github/workflows/*.yml, .github/
    ↪ workflows/*.yaml, .gitlab-ci.yml, .circleci/config.yml, azure-pipelines.yml,
    ↪ Jenkinsfile, etc.) - these often contain the most reliable setup steps and
    ↪ test commands
  b) If no CI/CD files exist or they're insufficient, check dependency/environment
    ↪ files (requirements.txt, setup.py, pyproject.toml, package.json, Gemfile,
    ↪ Cargo.toml, etc.)
  c) Then examine README files (README.md, README.rst, etc.) for setup instructions
  d) Look for other configuration files (Makefile, tox.ini, environment.yml, etc.)
3. Always start by browsing the repository directory structure, with particular focus
    ↪ on the priority order above.
4. The choice of test framework should be determined by the repository's contents and
    ↪ CI/CD configurations.
5. It is strictly prohibited to modify the test cases in the code repository.
6. Do not create a new Dockerfile for environment isolation.
7. Commands can be run without sudo as the current session already has root privileges.
    ↪
8. Based on the characteristics of the repository, it is possible to determine whether
    ↪  environment setup and test execution need to be performed within a virtual
    ↪ environment.
9. It may be necessary to install the repository from source before you can run code.
10. If you encounter package installation failures, try updating your local package
    ↪ index.
11. **Pay special attention to CI/CD matrix configurations** - they often test
    ↪ multiple Python/Node.js/etc. versions and can guide your environment setup.

GENERAL TIPS:
1. If you run a command and it doesn't work, try running a different command. A
    ↪ command that did not work once will not work the second time unless you modify
    ↪  it!
2. If you open a file and need to get to an area around a specific line that is not in
    ↪  the first 100 lines, say line 583, don't just use the scroll_down command
    ↪ multiple times. Instead, use the goto 583 command. It's much quicker.
3. Always make sure to look at the currently open file and the current working
    ↪ directory (which appears right after the currently open file). The currently
    ↪ open file might be in a different directory than the working directory! Note
    ↪ that some commands, such as 'create', open files, so they might change the
    ↪ current open file.
4. When using the edit command, always quote both search and replace arguments to
    ↪ avoid argument parsing failures.

INSTRUCTIONS:
Now, you're going to solve this issue on your own. Your terminal session has started
    ↪ and you're in the repository's root directory. You can use any bash commands
    ↪ or the special interface to help you. Edit all the files you need to and run
    ↪ any checks or tests that you want.
Remember, YOU SHOULD ALWAYS INCLUDE EXACTLY ONE TOOL CALL/FUNCTION CALL PER RESPONSE.
When you're satisfied with all of the changes you've made, you can submit your changes
    ↪  to the code base by simply running the submit command.
Note however that you cannot use any interactive session commands (e.g. python, vim)
    ↪ in this environment, but you can write scripts and run them. E.g. you can
    ↪ write a python script and then run it with the python command.
```

```
next_step_template: |-
{{observation}}
(Open file: {{open_file}})
(Current directory: {{working_dir}})
bash-$
next_step_no_output_template: |-
Your command ran successfully and did not produce any output.
(Open file: {{open_file}})
(Current directory: {{working_dir}})
bash-$
success_reflection_template: |-
You are an expert code assistant who just successfully repaired a software project in
    ↪ an automated programming session. Your task now is to reflect on this
    ↪ successful task and summarize key takeaways for future similar tasks.

CONTEXT:
Below is the full interaction history during the session, including commands issued,
    ↪ file edits, and observations returned. Use this history to extract useful
    ↪ insights.

----------------
{{session_history}}
----------------

PLEASE WRITE A REFLECTION INCLUDING:
1. **Problem Summary**:
  - What was the main problem or failure mode encountered?

2. **Critical Fix Steps**:
  - List 2-5 concrete steps that were essential to achieving success.
  - For each step, describe what the agent did and why it was effective.

3. **Heuristics or General Patterns**:
  - Are there any general rules or repeatable strategies that could help in future
    ↪ repairs?

4. **Environment or Tooling Insights** (if applicable):
  - Any key observations about the build system, environment configuration, or command-
    ↪ line tools that helped?

Here is the traj:
```

Listing 2. Prompt of Extract Codebase-related Info

```
As an expert software engineer, your task is to analyze project files to extract the
    ↪ technical components required to set up its environment.
**IMPORTANT CONTEXT: You are operating inside a fresh, clean Linux container (e.g.,
    ↪ Ubuntu22.04). Your goal is to determine the necessary commands and tools to
    ↪ install *directly within this existing container*.**

**CRITICAL CONSTRAINTS:**
```

```
- **DO NOT include Docker-related setup steps.** You are already inside a container,
    ↪ so ignore Dockerfile, docker build, docker run, or any containerization
    ↪ instructions.
- **Focus on what needs to be installed INSIDE the container**, not how to build or
    ↪ run containers.
- If you see a Dockerfile, extract the `RUN apt-get install ...` commands as system
    ↪ dependencies, but ignore `FROM`, `docker build`, or `docker run` instructions.

--- README.md ---
{readme}
--- END README.md ---

--- Workflow Guidelines (e.g., .github/workflows/ci.yml, Dockerfile) ---
{workflow_guide}
--- END WORKFLOW GUIDELINES ---

Based on the files, extract the key information into the following JSON schema.

**CONSTRAINTS:**
- **DO NOT include Docker-related setup steps.** Your analysis should focus on what
    ↪ needs to be done *inside* a container, not on how to build one. For example,
    ↪ if you see a Dockerfile, extract the `RUN apt-get install ...` commands as
    ↪ system dependencies, but ignore `FROM`, `docker build`, or `docker run`
    ↪ instructions.
- If a tool or dependency is mentioned, extract it. If not, use `null` or an empty
    ↪ list `[]`.

Fill the following JSON schema:
{{
    "language": {{
        "name": "string (e.g., Python, Node.js, C/C++)",
        "version_constraint": "string (e.g., 3.10, >=16.x)"
    }},
    "package_manager": {{
        "name": "string (e.g., pip, npm)",
        "install_command": "string (e.g., pip install -r requirements.txt)",
        "config_file": "string (e.g., pyproject.toml, package.json)"
    }},
    "build_tools": ["string (e.g., make, cmake, gradle)"],
    "test_framework": ["string (e.g., pytest, jest)"],
    "system_dependencies": {{
        "identified_in_ci": ["string (system packages from CI/workflow files, e.g.,
    ↪ gcc, make)"],
        "likely_required": ["string (inferred but not explicitly listed system
    ↪ dependencies, e.g., libssl-dev)"]
    }},
}}
Return only the JSON object with the extracted information.
```

Listing 3. Prompt of Generate Hypothetical Query

```
    As an experienced developer setting up a project, generate 3-5 hypothetical,
    ↪ specific questions you might ask when facing setup problems. Base your
    ↪ questions on the provided technical analysis. These questions should be
    ↪ perfect for retrieving solutions from a knowledge base.

    **IMPORTANT CONTEXT: You are setting up the environment INSIDE an existing Linux
    ↪ container. DO NOT include Docker-related questions or containerization setup.**
    ↪

    --- Project Technology Stack Analysis ---
    {profile_str}
    --- END ANALYSIS ---

    The questions must be:
    1.  INTENT-ALIGNED: Address specific problems (e.g., "How to fix...").
    2.  CONTEXT-RICH: Include versions, OS, libraries from the analysis.
    3.  PROACTIVE: Predict common pitfalls like version conflicts or missing
    ↪ dependencies.
    4.  CONTAINER-AWARE: Focus on installing tools and dependencies inside the
    ↪ container, not on containerization itself.

    **AVOID questions about:**
    - Docker setup, container building, or containerization
    - "How to run this in Docker" or "How to build a container"
    - Container orchestration or deployment

    **FOCUS on questions about:**
    - Installing system packages with apt-get/yum
    - Setting up language runtimes and package managers
    - Configuring build tools and dependencies
    - Resolving version conflicts within the container

    Return only a JSON object with a "queries" key containing a list of question
    ↪ strings.
```

Listing 4. Prompt of Experience Extraction from Trajectories

```
You are an expert in analyzing environment configuration problems, specializing in
    ↪ identifying and analyzing environment-related issues from agent trajectories.

## Objective
Based on the given agent trajectory, use **Chain of Thought** analysis process to
    ↪ accurately identify environment configuration problems and extract structured
    ↪ solutions.

## Analysis Steps
1. **Problem Identification**: Carefully read the trajectory to identify all
    ↪ environment configuration-related failures or errors
2. **Root Cause Analysis**: Deeply analyze the root cause of each problem
3. **Solution Extraction**: Extract actually effective solutions from the trajectory
4. **Structured Output**: Generate standardized triple format
```

```
## Key Requirements
- **Generality**: Descriptions should be **repository-agnostic**, focusing on general
    ↪ environment configuration problem types rather than being specific to a
    ↪ particular project
- **Completeness**: Ensure all environment configuration-related problems are
    ↪ identified and recorded
- **Accuracy**: Only record problems that actually occurred in the trajectory and
    ↪ solutions that were truly effective

## Output Format
Return a valid JSON object with the following structure:
{
    "triples": [
        {
            "problem": "Concise problem description (e.g., 'The initial make command
    ↪ failed when building the project')",
            "solution": "Detailed solution explanation including root cause analysis (
    ↪ e.g., 'The make tool was not installed on the Ubuntu system, which is required
    ↪  for building projects using Makefiles. The root cause was a missing build
    ↪ tool package in the environment configuration. The solution was to install
    ↪ make using apt-get install -y make')",
            "action": "The specific command used to solve the problem (e.g., 'apt-get
    ↪ install -y make')"
        }
    ]
}
**IMPORTANT**: Return ONLY valid JSON format, no additional text or explanations.
```

S2 Initial Process of our TDM

The initial process of our Trajectory-Derived Memory (TDM) primarily focuses on constructing a robust and generalizable knowledge base. This stage is crucial for populating the TDM with diverse and high-quality experiential data derived from real-world software engineering scenarios. As outlined in Section 3.2.2, the raw data for TDM originates from the execution trajectories of agents performing Automated Execution and Correction (AEC) tasks. Specifically, these trajectories are generated by agents interacting with a diverse set of open-source projects hosted on GitHub. These projects serve as the foundational source for our TDM's initial knowledge base, providing a rich array of software engineering challenges encountered during development and maintenance.

Table S1 lists the selected GitHub repositories from which these AEC task instances are derived for the TDM's initial stage. This deliberate selection is critical for several reasons. Many of these repositories are part of, or representative of, the challenging software engineering scenarios provided by benchmarks like Multi-SWE-Bench. By including projects spanning multiple programming languages (e.g., Rust, JavaScript, Java, Go, C++), diverse frameworks, and various dependency managers, we ensure that the collected execution trajectories offer a broad and representative sample of real-world AEC problems. This foundational diversity is paramount for building a TDM whose extracted experiential knowledge is both robust and broadly generalizable, enabling it to provide effective decision support across a wide spectrum of novel challenges.

Table S1. Selected AEC Task Instances for TDM's Initial Stage.

Column1	Column2	Column3
alibaba/fastjson2	darkreader/darkreader	googlecontainertools/jib
anuraghazra/github-readme-stats	elastic/logstash	grpc/grpc-go
apache/dubbo	expressjs/express	iamkun/dayjs
axios/axios	facebook/zstd	jqlang/jq
BurntSushi/ripgrep	facebookresearch/hydra	Kong/insomnia
catchorg/Catch2	fasterxml/jackson-core	mockito/mockito
clap-rs/clap	fasterxml/jackson-databind	mui/material-ui
cli/cli	fasterxml/jackson-dataformat-xml	nlohmann/json
conan-io/conan	fmtlib/fmt	nushell/nushell
ponylang/ponyc	python/mypy	rayon-rs/rayon
serde-rs/serde	sharkdp/bat	sharkdp/fd
simdjson/simdjson	sveltejs/svelte	tokio-rs/bytes
tokio-rs/tokio	tokio-rs/tracing	vuejs/core
yhirose/cpp-httplib	zeromicro/go-zero	/

S3 Human Expert Cross-Review

In the experimental section of the main paper, we validated the proposed hybrid evaluation mechanism by conducting a systematic cross-review with three human experts. Here, we provide additional details on the review process and the construction of the evaluation dataset.

The three experts involved in this study are the authors **Hanwu Chen**, **Hanyu Lin**, and **Zhanjiang Yang**. Each expert was tasked with independently reviewing the execution trajectories produced by the agent when performing automatic environment configuration (AEC) tasks. These trajectories capture the sequence of observations, thoughts and actions generated by the agent. By using diverse tasks collected from Multi-SWE-Bench, the evaluation directly reflects the complexity and variability encountered in practical software engineering scenarios.

To ensure objectivity, each expert assessed the success or failure of the agent's trajectory without access to the judgments of the other reviewers. The evaluation criteria focused on whether the trajectory ultimately led to a comprehensive test report, while also considering the plausibility and coherence of the intermediate steps. After the individual reviews were completed, the final decision for each task instance was determined through a majority vote. This procedure not only mitigates individual biases but also enhances the robustness and credibility of the evaluation results.

Table S2 summarizes the 83 unique task instances selected for this human expert cross-review. These instances were carefully de-duplicated to provide a clean and representative dataset. The collection spans a wide range of programming languages, frameworks, and dependency management ecosystems, including projects in C/C++, Rust, Go, Python, Java, JavaScript/TypeScript and PHP. This diversity was deliberately ensured so that the expert evaluation would cover heterogeneous software environments, reflecting the generalizability of the proposed method.

By combining independent expert judgments with a principled voting mechanism, this cross-review serves as a reliable "gold standard" against which the hybrid evaluation approach can be benchmarked. It demonstrates that the proposed mechanism not only aligns with expert assessments but also scales more efficiently to large numbers of tasks, thereby providing both accuracy and practicality in real-world evaluation settings.

Table S2. List of 83 Task Instances Evaluated by Human Experts.

Org	Repo	PR Numbers
alacritty	alacritty	8192
apache	shenyu	5869
astropy	astropy	13033
Automattic	mongoose	14692, 15302
Automattic	wp-calypso	101046
beego	beego	5725
briannesbitt	Carbon	3170
caddyserver	caddy	6669
caolan	async	1224, 1790
CGAL	cgal	8736
checkstyle	checkstyle	15448
colinhacks	zod	3887
composer	composer	12367
concourse	concourse	9103
django	django	10097
doctrine	dbal	6890
facebook	lexical	6834
facebook	react	30951
fatedier	frp	3664
fish-shell	fish-shell	10472
fluent	fluent-bit	3663, 10007
gin-gonic	gin	4048
go-gorm	gorm	5974
gohugoio	hugo	13495
google	zx	811, 1113
Graylog2	graylog2-server	14865, 22093
halide	Halide	5545, 6533, 7506, 8490
helix-editor	helix	6675
istio	istio	55229
jesseduffield	lazygit	4369
junit-team	junit5	3394, 4366
junegunn	fzf	4231
KaTeX	KaTeX	3735
labstack	echo	2717
laravel	framework	55507
libsdl-org	SDL	11946
matplotlib	matplotlib	13989
MetaMask	metamask-extension	30527
micronaut-projects	micronaut-core	11440
mruby	mruby	3649, 6442
mwaskom	seaborn	3187
nasa	openmct	7986
nuxt	nuxt	24511, 31081

Org	Repo	PR Numbers
OpenMathLib	OpenBLAS	4729
OpenRefine	OpenRefine	7174
pallets	flask	5014
palantir	blueprint	6882
payloadcms	payload	12489
php	php-src	17835
provectus	kafka-ui	3505
psf	requests	1142
pydata	xarray	4629
pylint-dev	pylint	4970
pytest-dev	pytest	5262
react-hook-form	react-hook-form	12793
redis	redis	13711
reduxjs	redux	4519
rjsf-team	react-jsonschema-form	4637
root-project	root	17731
rust-lang	mdBook	2486
scikit-learn	scikit-learn	10297
slimphp	Slim	3319
sphinx-doc	sphinx	10673
sympy	sympy	12096
syncthing	syncthing	9914
swagger-api	swagger-ui	10390
tj	commander.js	987, 1926
trpc	trpc	5532, 5850
valkey-io	valkey	1694