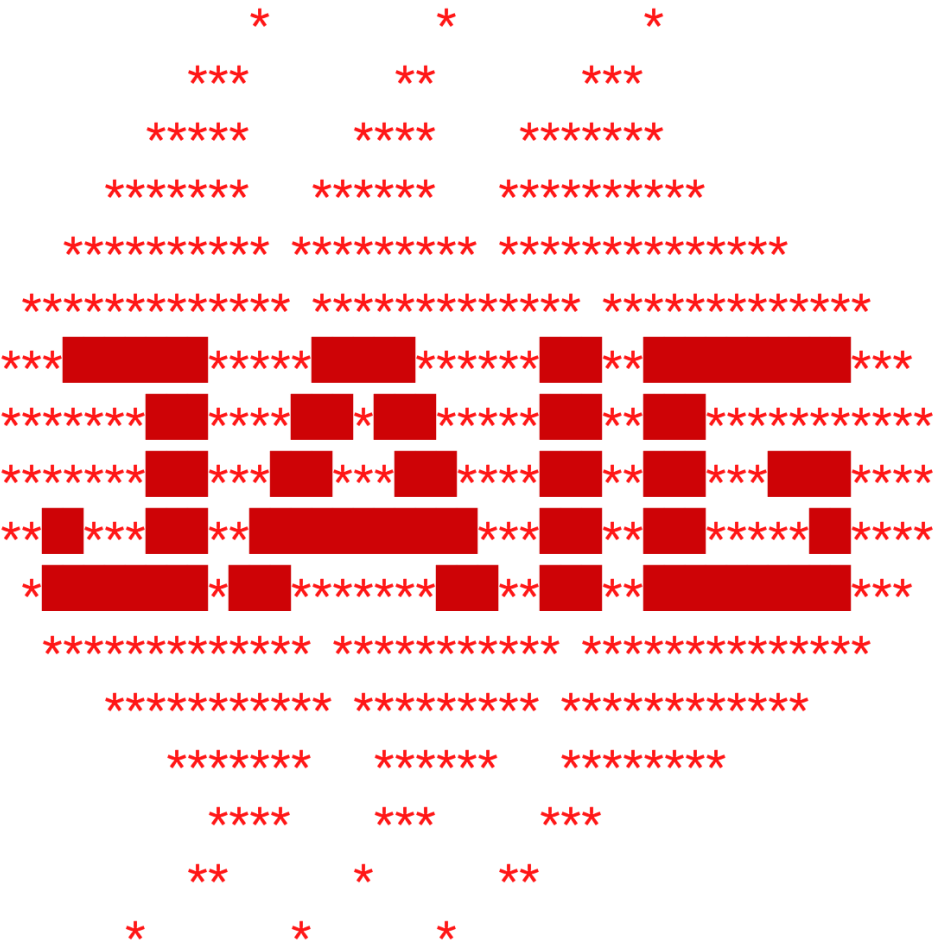
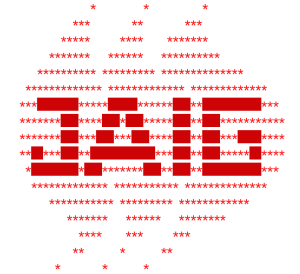


Java AI-powered Code Generator



Working with the Code Generator

- 1) Create a folder
- 2) Place the request there in a file with a .txt extension (for example, **prompt.txt**)
- 3) Start generation by pressing JAIG button




Prompt Privacy: the importance of API

Prompt security: 3 options

Chat GPT => insecure

Open AI **API** => **secure**

Local LLM like llama => completely secure

 Research ▾ API ▾ ChatGPT ▾ Safety Company ▾ Search Log in ↗ Try ChatGPT ↗

Enterprise privacy at OpenAI

Trust and privacy are at the core of our mission at OpenAI. We're committed to privacy and security for ChatGPT Enterprise and our API Platform.

[Read commitments ↓](#)

Our commitments

Ownership: You own and control your data

- ✓ We do *not* train on your data from ChatGPT Enterprise or our API Platform
- ✓ You own your inputs and outputs (where allowed by law)
- ✓ You control how long your data is retained (ChatGPT Enterprise)

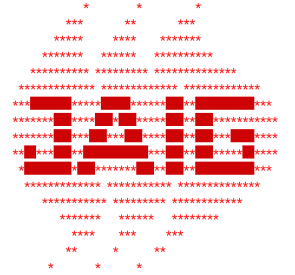
Control: You decide who has access

- ✓ Enterprise-level authentication through SAML SSO
- ✓ Fine-grained control over access and available features
- ✓ Custom models are yours alone to use, they are not shared with anyone else

Security: Comprehensive compliance

- ✓ We've been audited for SOC 2 compliance
- ✓ Data encryption at rest (AES-256) and in transit (TLS 1.2+)
- ✓ Visit our Trust Portal to understand more about our security measures

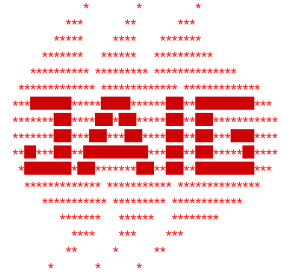
Working with the Code Generator



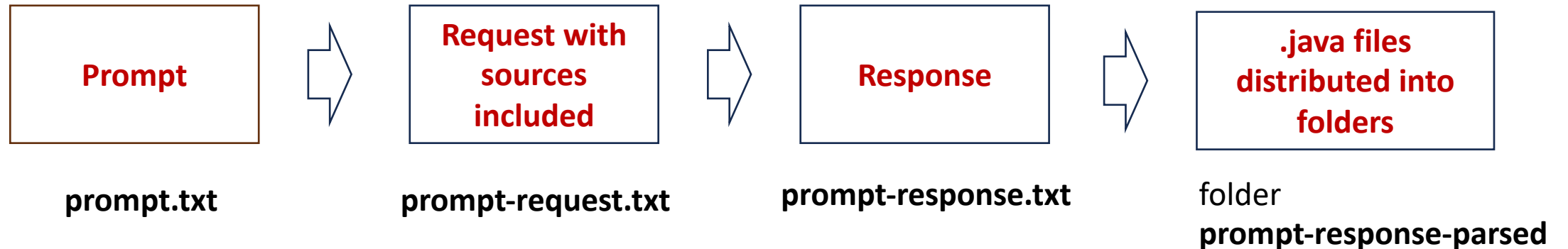
- 4) Add **references** to the files or folders to the prompt
- 5) The contents of the files are added to the Request



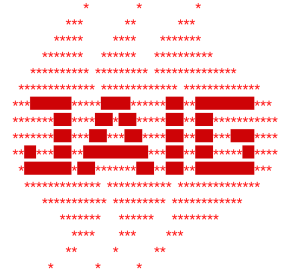
Working with the Code Generator + parsing



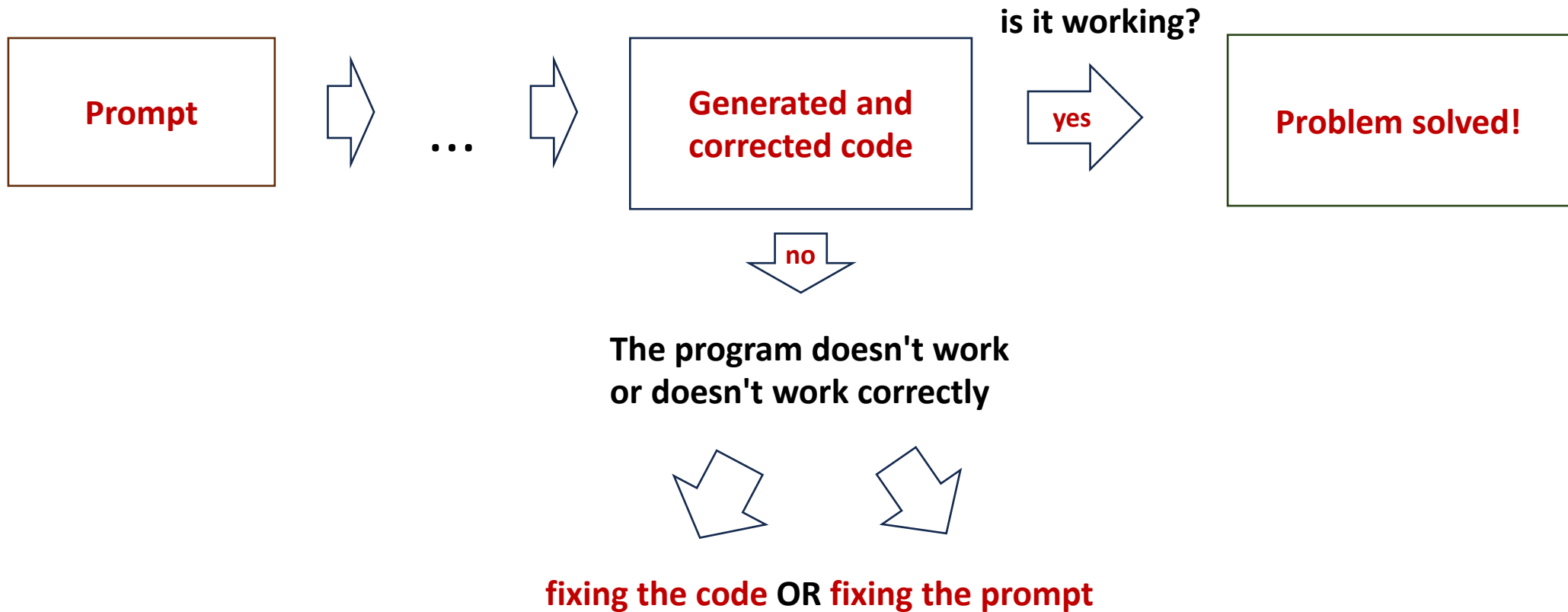
- 6) If the response contains code, and this code has packages specified (number of packages = number of classes and interfaces), **automatic parsing** of the response is performed: files are distributed into folders according to the package
- 7) The result of the parsing is written to the **prompt-response-parsed** folder



Working with the Code Generator: Testing



Now, the code needs to be executed and tested

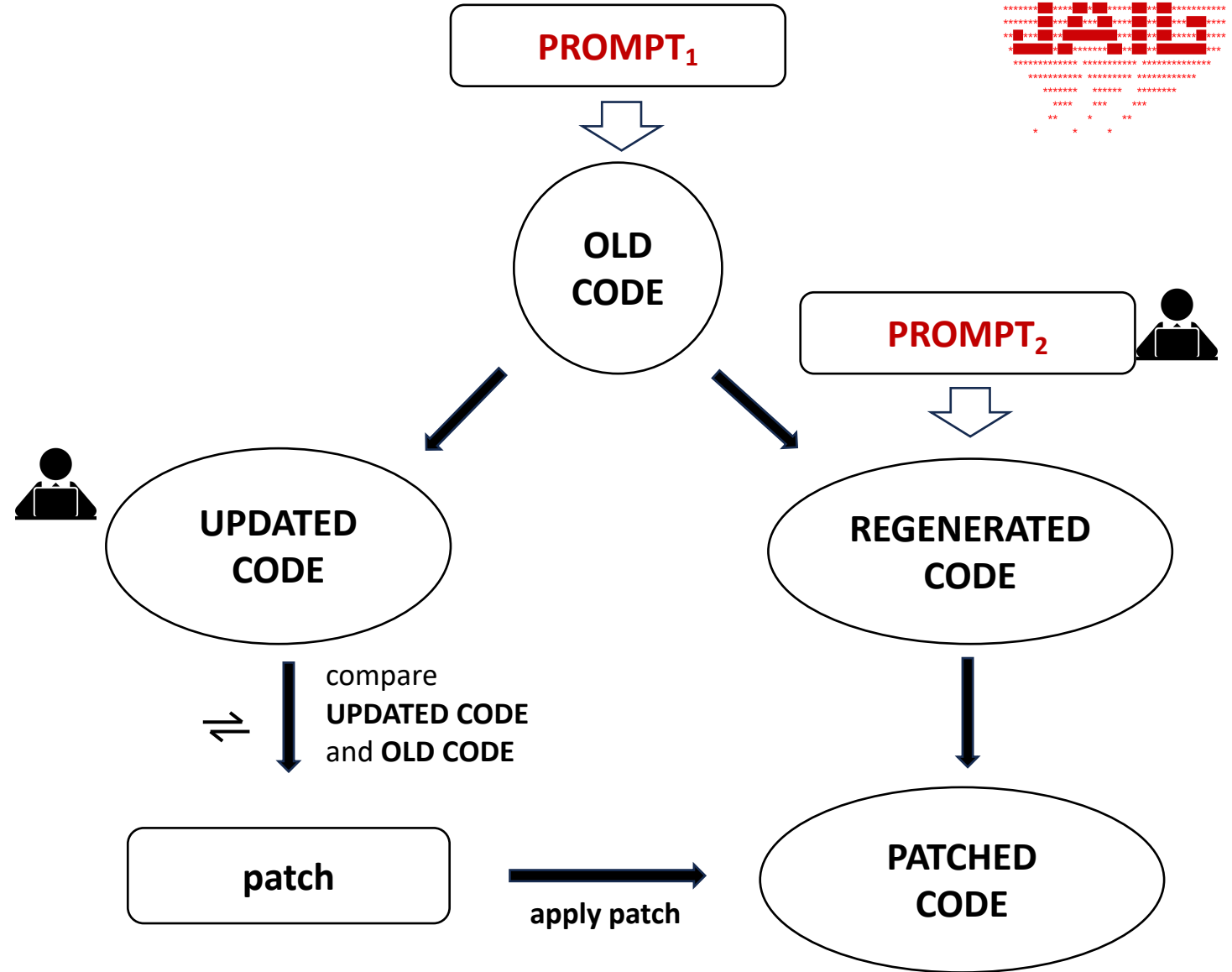


Patch GPT response

We can have
2 types of changes:

- In the **code**
- In the **prompt**

If you have **both**, you need to
merge them



Merge code and GPT response

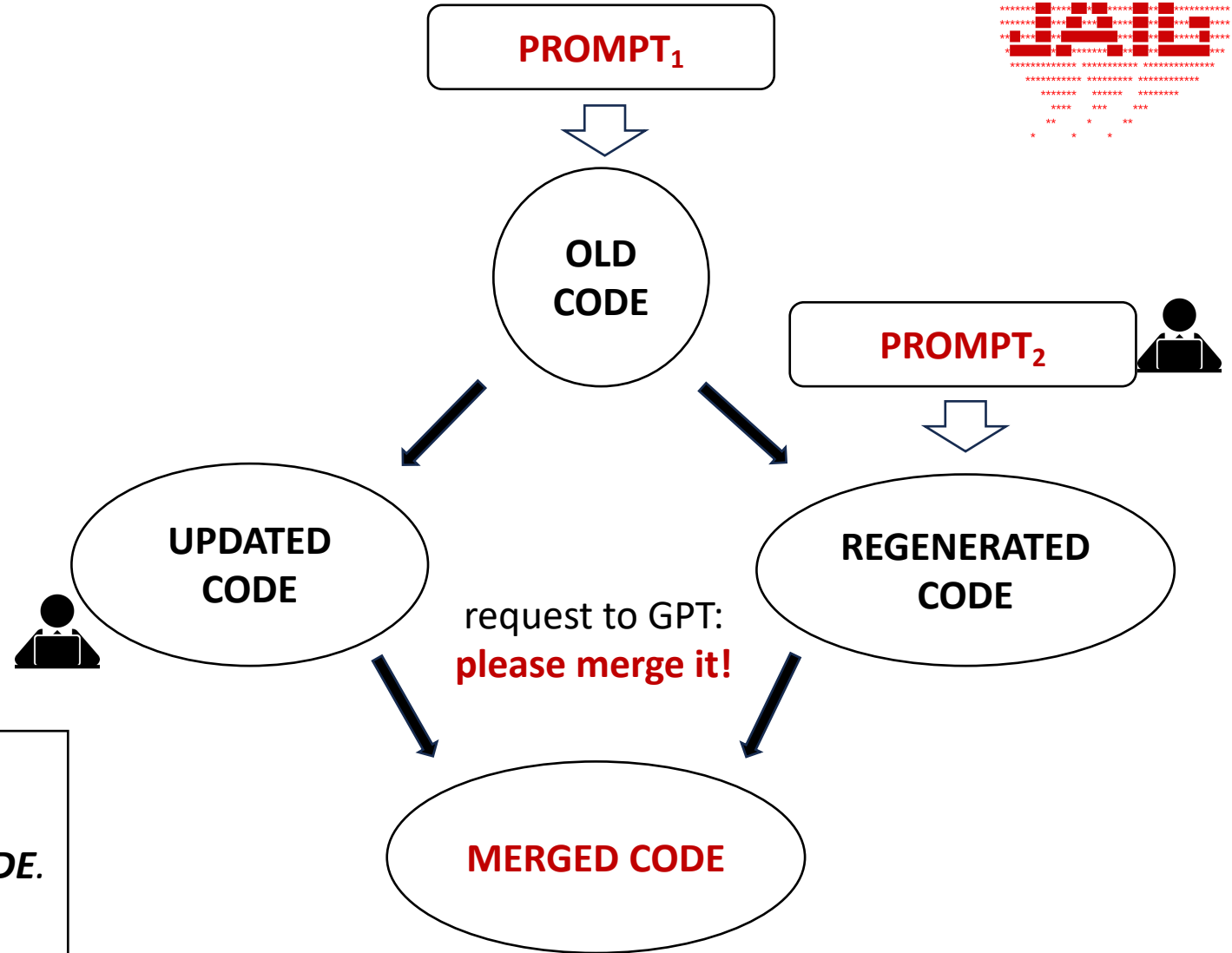
We can have
2 types of changes:

- In the **code**
- In the **prompt**

If you have **both**, you need to
merge them

merge prompt be like:

*Dear Chat GPT,
I had **OLD CODE**.
Then programmer changed it to **UPDATED CODE**.
And you changed it to **REGENERATED CODE**.
Can you please **merge all** these changes?*



Patching or merging?

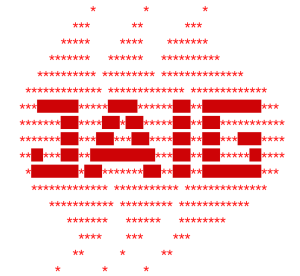
Patching

- **Patching is faster** and doesn't require an AI generation
- Patching can be **precisely controlled**
- For patching, we can open the **<prompt>.patch** file and see exactly what has been changed
- Works well for simple cases, but **fails on more complicated**

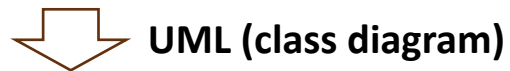
Merging

- For merging, we're **asking the AI** to merge the changes => it is **slower** and more **expensive**
- The work of merging is **less reliable** and cannot be controlled
- Merging is better suited in **complex cases** when you need to merge non-obvious changes in the code and in the generation results

Application Generation Scheme



Domain Study and Domain Model Development



Implementing the Domain Model in Java (Basic)



Spring Data / JPA persistence

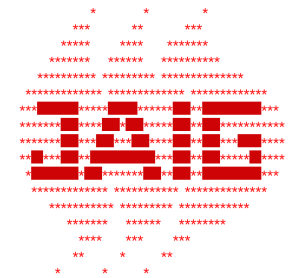


Implementation of REST API (Spring REST controllers, tests)



Natural Language Interface Development

- ✓ requests
 - > 1_requirements
 - > 2_requests-basic
 - > 3_requests-jpa
 - > 4_requests-rest
 - > 5_requests-NL



Spring Application Generation Scheme

Domain Study and Domain Model Development

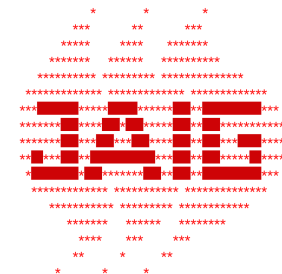
- Generate use cases
- Approve use cases with the customer
- Domain Model Generation
- Approval of the domain model with the customer

Implementing the Domain Model in Java (Basic)

Spring Data / JPA persistence

Implementation of REST API (Spring REST controllers, tests)

Natural Language Interface Development



Spring Application Generation Scheme

Domain Study and Domain Model Development

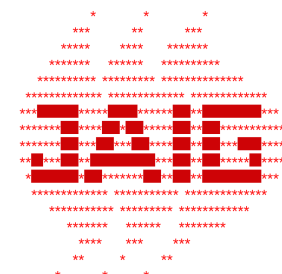
Implementing the Domain Model in Java (Basic)

- Generation of a domain model in Java based on UML diagrams
- Distributing Classes to Packages
- Validate the domain model by populating it with test data
- Validating the Domain Model by Generating a Simple UI

Spring Data / JPA persistence

Implementation of REST API (Spring REST controllers, tests)

Natural Language Interface Development



Spring Application Generation Scheme

Domain Study and Domain Model Development

Implementing the Domain Model in Java (Basic)

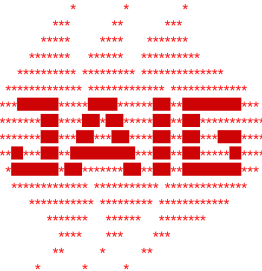
Spring Data / JPA persistence

- Adding **JPA** annotations
- Adding Spring Data **repositories**
- Verifying that Test Data Is **Being Saved**
- Troubleshooting Potential Issues

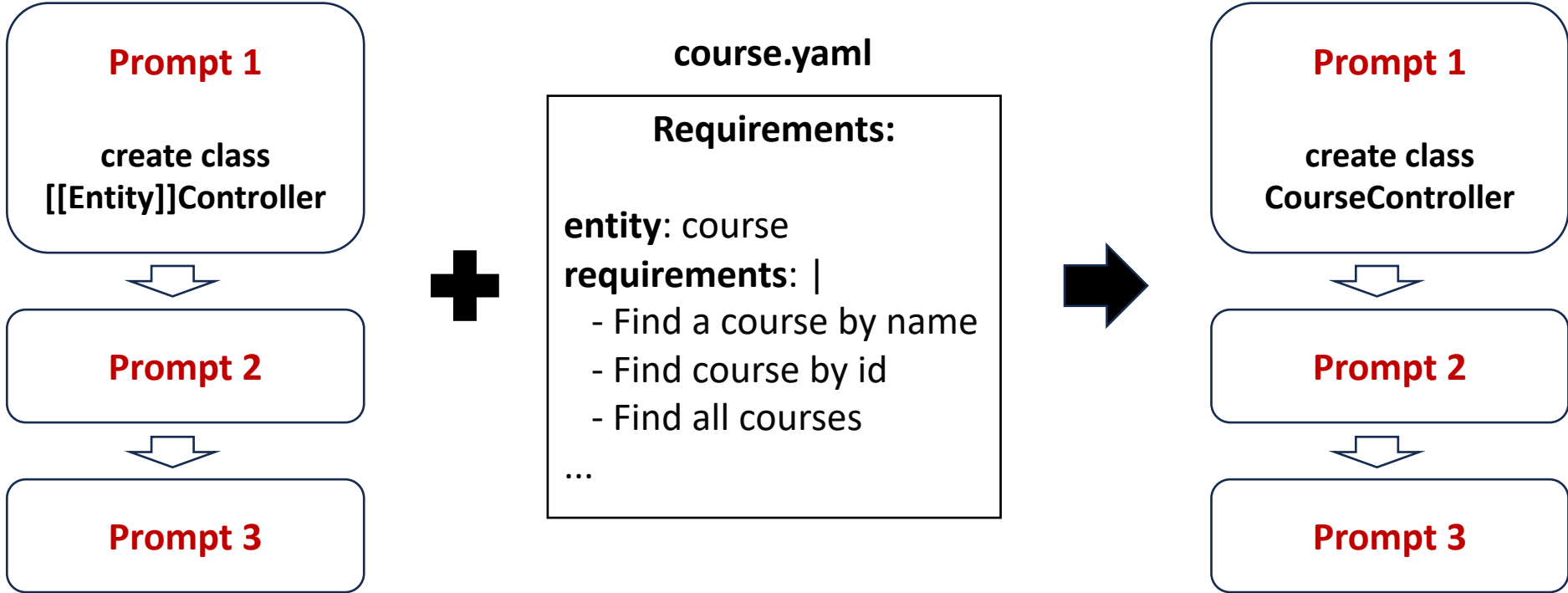
Implementation of REST API (Spring REST controllers, tests)

Natural Language Interface Development

JAIG Templates

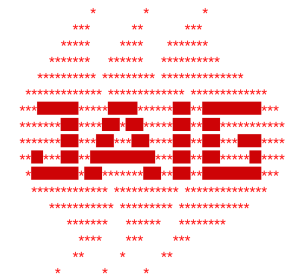


Template prompts



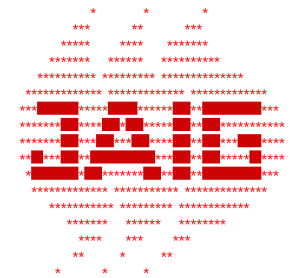
entity: course

- **Entity:** Course
- **entities:** courses
- **Entities:** Courses



Spring Application Generation Scheme

Domain Study and Domain Model Development
Implementing the Domain Model in Java (Basic)
Spring Data / JPA persistence
Implementation of REST API (Spring REST controllers, tests) <ul style="list-style-type: none">• Generation of REST APIs according to requirements and domain model• Approval of the REST API by the customer (contract first), modification if necessary• Generating Tests for REST APIs• Generation of Spring controllers and services• Running and testing the code• Generating Unit Tests for Controllers• Generation of Open API documentation (Swagger)
Natural Language Interface Development



Spring Application Generation Scheme

Domain Study and Domain Model Development

Implementing the Domain Model in Java (Basic)

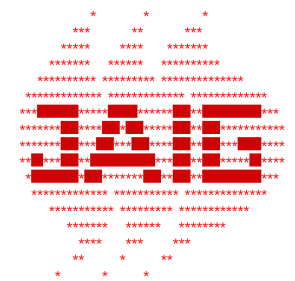
Spring Data / JPA persistence

Implementation of REST API (Spring REST controllers, tests)

Natural Language Interface Development

- Generating a client for a JavaScript REST API
- Turn a natural language query into JavaScript code
- SQL Generation
- Creating a GPT Agent

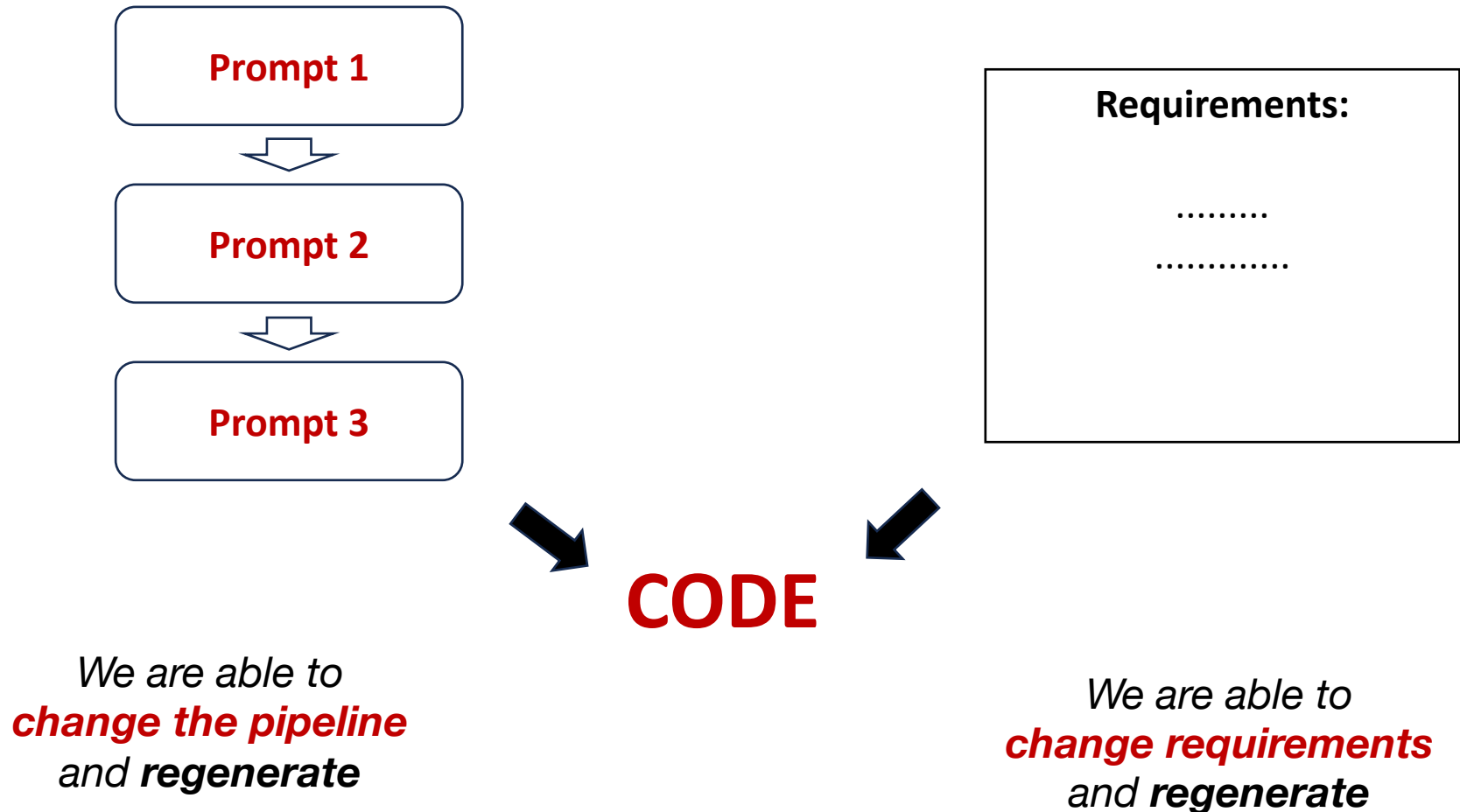
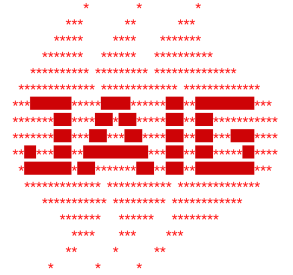
The main features of JAIG are:



- Sending requests to the **GPT API**
- **Including source files** into request
- Seamless **integration** with the development environment (**IDEA**)
- **View requests during the generation process**, code highlighting
- **Parsing** of generation results, sorting of source files and packages
- **Writing** generated files to **src/main/java**
- Ability to write results to an arbitrary folder (**#save-to**)
- **Flexible** generation **settings**, including **model** selection (GPT3/GPT4 and others) and **temperature**, globally or on a per-request basis
- Creating and applying **patches** for code changes
- **Merging changes** in the code and in the results of code generation by means of AI
- Generating a series of queries from **templates**
- Execution of a series of queries (**batches**)
- **Rollbacks**
- **Refactoring** the selected code fragment using AI in dialog mode, using the prompt library

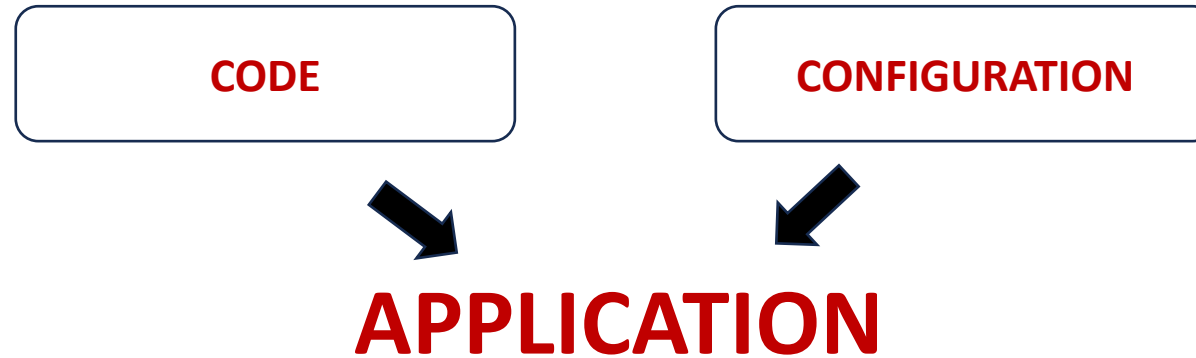
Development cycle with JAIG:

- Create a universal **prompts pipeline** of application generation
- Apply any **business domain** to generate a standardized code

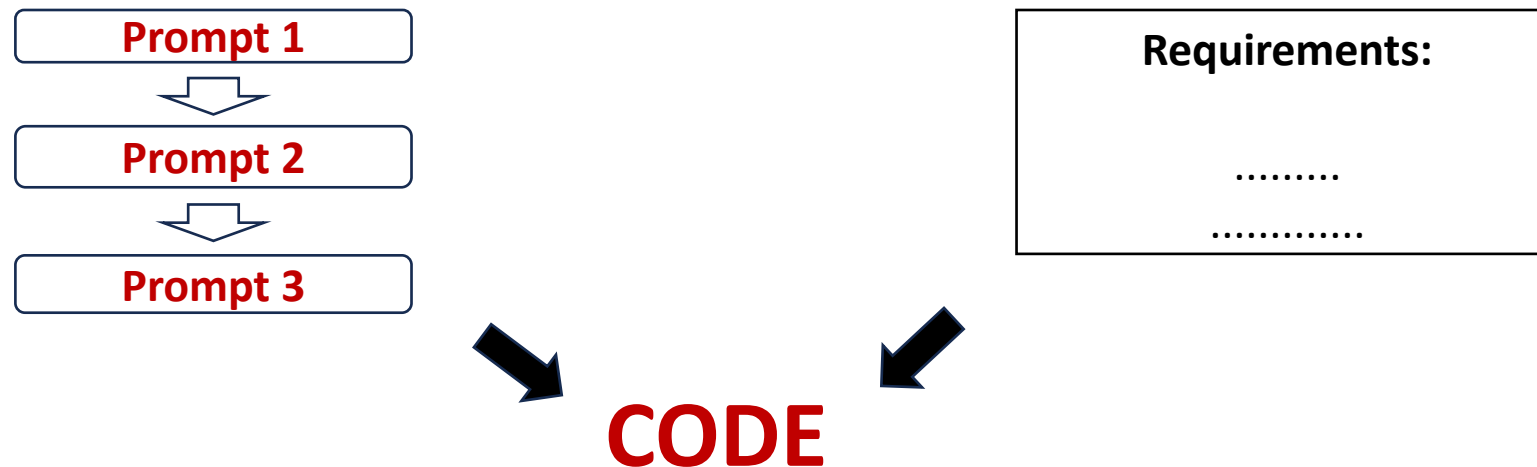


Spring and JAIG

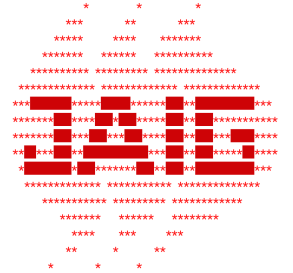
Spring separates **code** from **configuration**



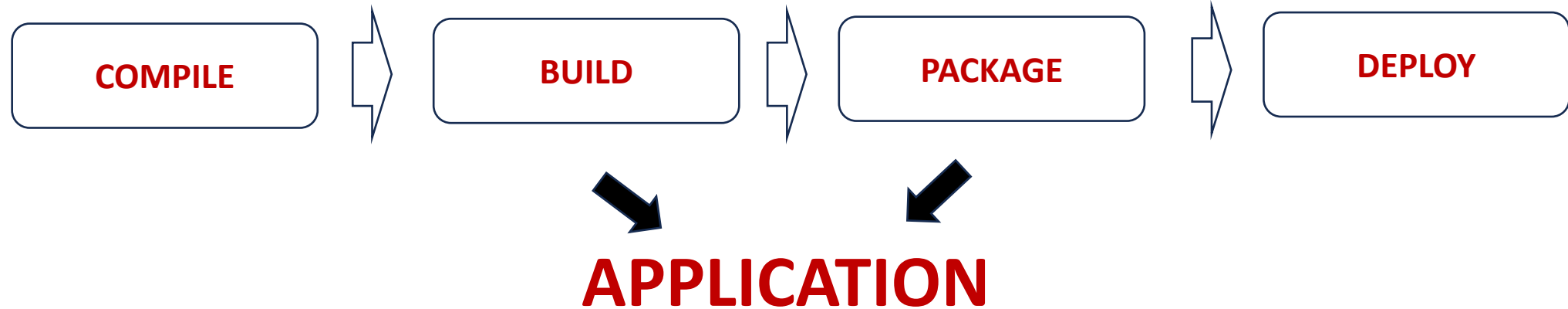
JAIG separates **prompt pipeline** to implement code (specific for your project) from **requirements**



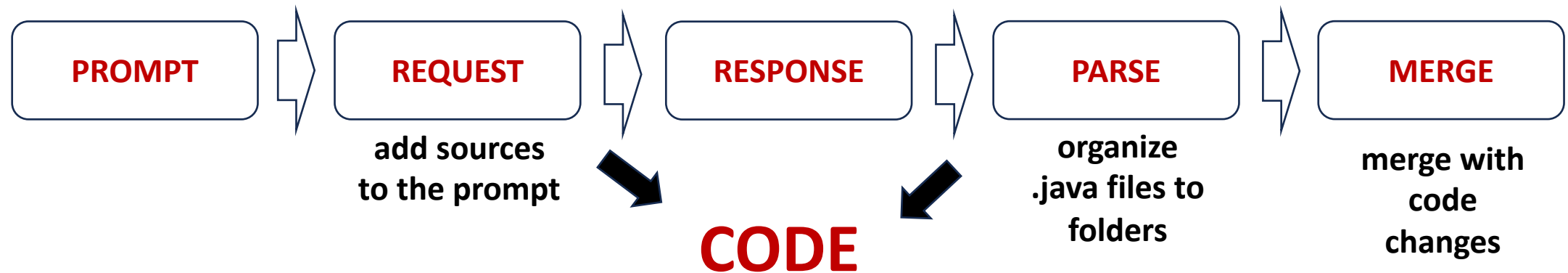
Maven and JAIG



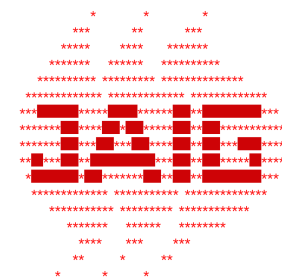
Maven defines **Life Cycle** to build & deploy the **Application**



JAIG defines **Life Cycle** to create a **Code** for Application



Copilot and JAIG



Natural Language Queries

How to connect GPT to you system:

- Natural Language to Java REST Client
- Natural Language to Java Services layer
- Natural Language to SQL
- Natural Language to JavaScript

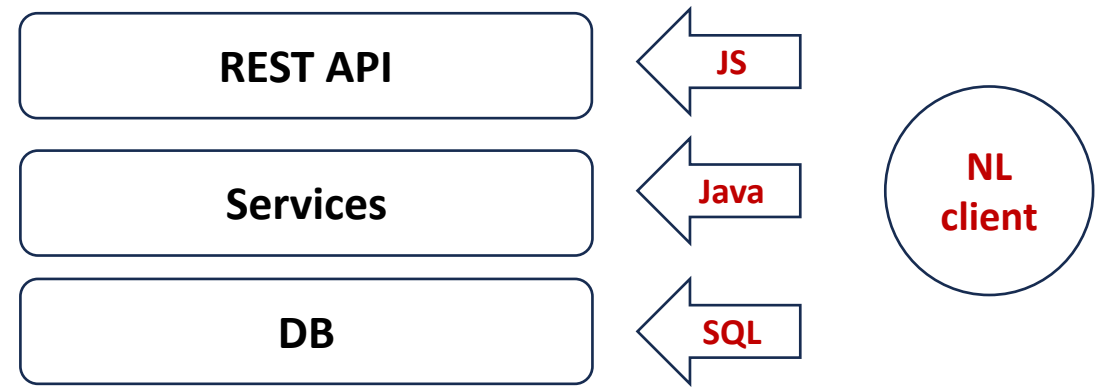
Cons SQL:

- The Dangers of SQL Injection (similar injection may happen for any type of interface – if you provide interface, it can be accessed via NL request)
- Ability to provide data that should not be visible
- Limited validation capabilities (only via constraints)

Benefits of SQL:

- Maximum flexibility for complex queries
- Brevity (Fast Generation)
- Maximum Performance

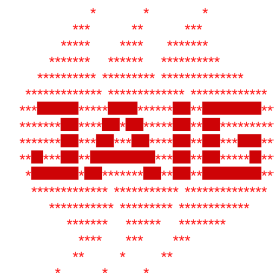
We can use NL requests to access different layers of our app:



Benefits of using JavaScript for Natural Language queries:

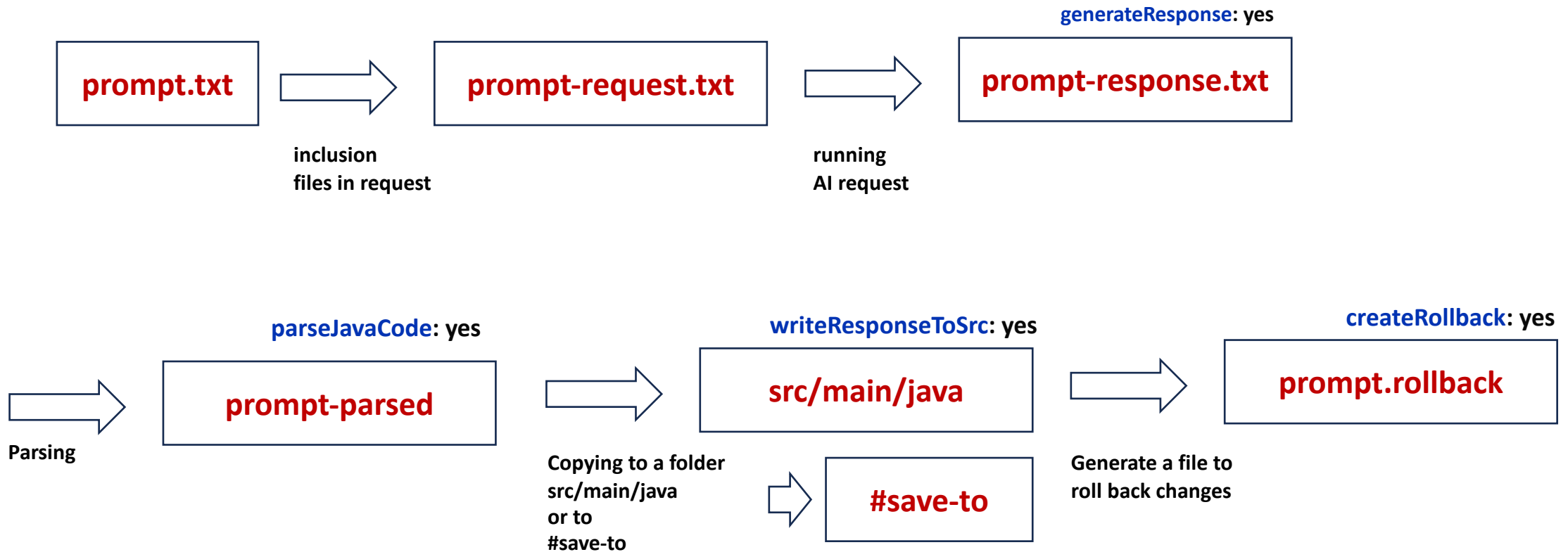
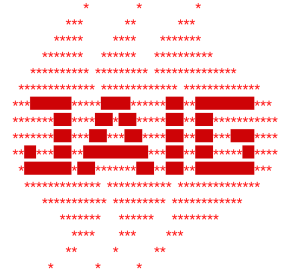
- No typing (less code generated)
- Code Brevity (Fast Generation)
- Can be integrated into the browser
- The Most Dynamic Option

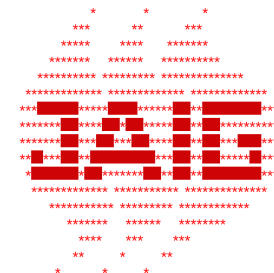
Disadvantage: integration is only possible via REST API



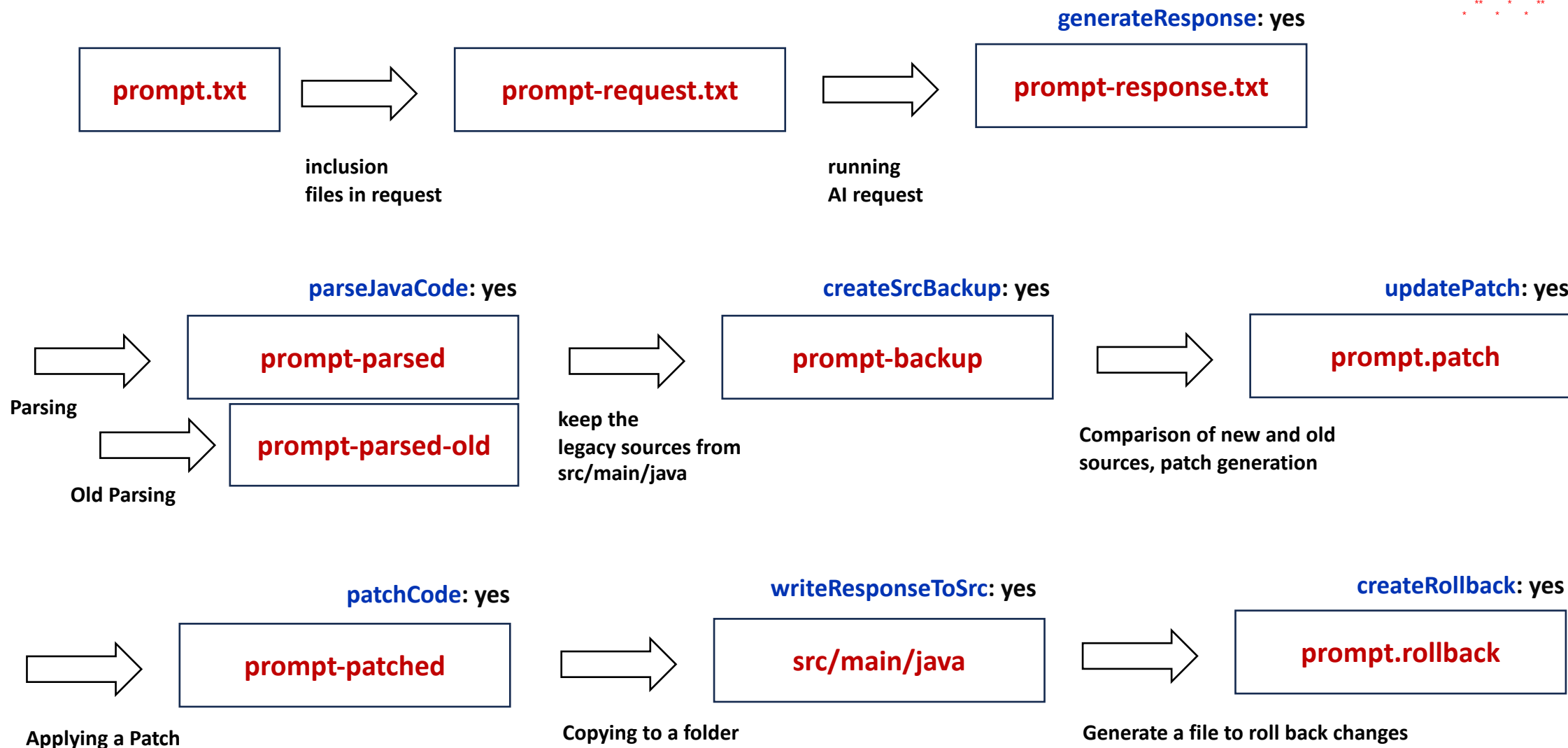
JAIG Brief Reference

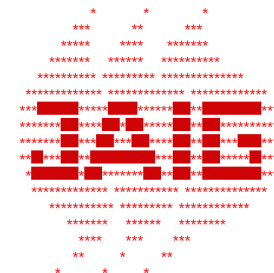
JAIG Lifecycle: generated artifacts



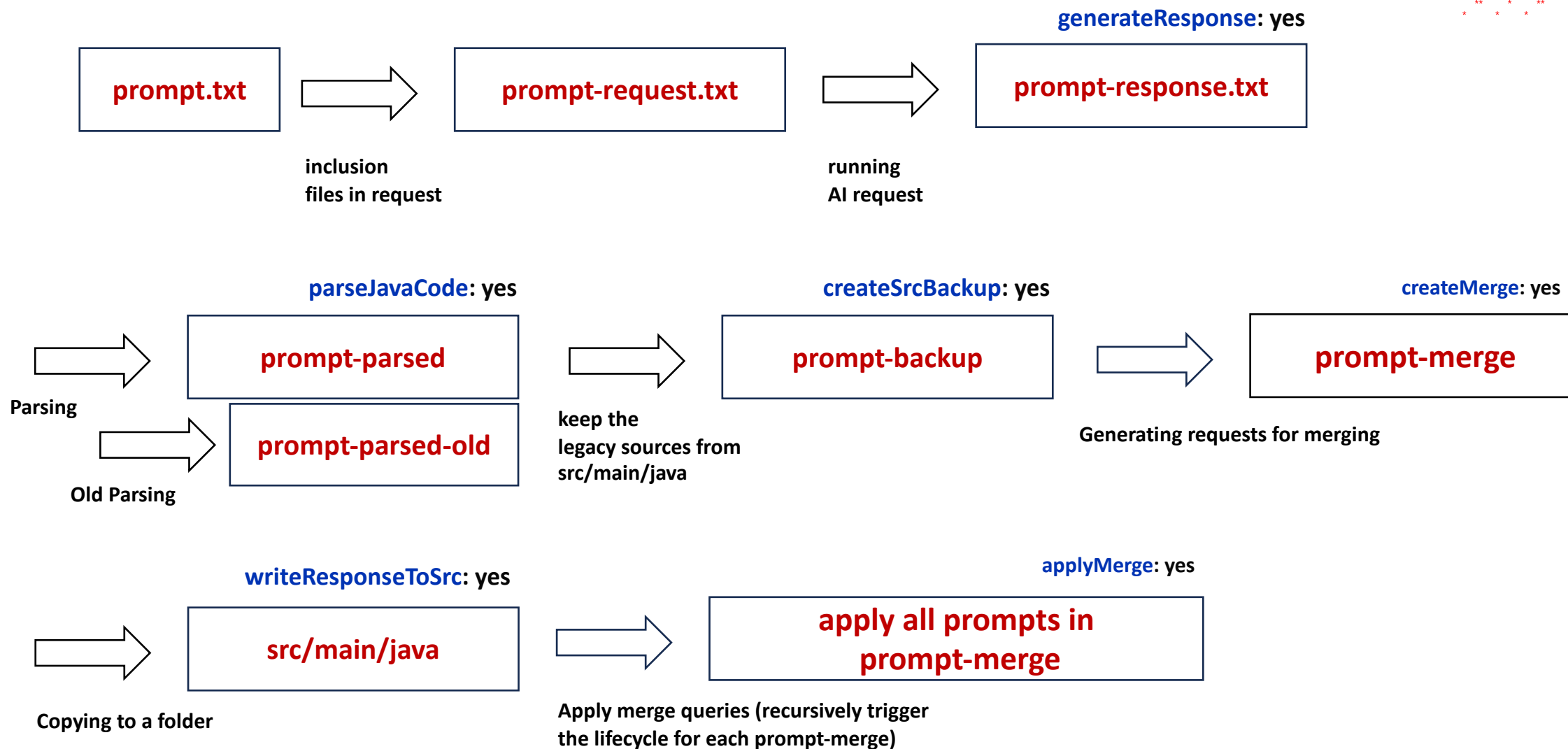


JAIG Lifecycle with Patching: generated artifacts

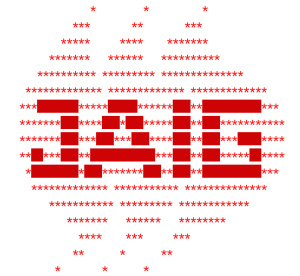




JAIG Lifecycle with Merging: generated artifacts



JAIG can be applied to:



To apply JAIG, open the file or focus on file/folder in Project Tree, then press JAIG button

.txt file -> send **prompt** to GPT and start the **JAIG lifecycle**

FOLDER -> find all .txt files, **create a .batch** with all prompts found in subfolders

OR **cleanup folder** (remove all generated files/folders)

FOLDER like NN_smth (e.g. 05_smth) -> insert the folder and shift all the following folders down

.yaml file with the specified template (e.g. template: templates/rest)

-> read the **requirements** and **create prompts** from the **template**

.batch file -> treat each line as a prompt and **execute sequentially**

.patch file -> **apply the patch** to the -parsed folder, save the result to -patched

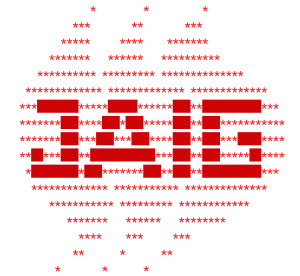
-parsed.rollback file -> apply rollback by removing generated files or restoring files to the original state

-full.rollback file -> roll back all changes made by this prompt

-response.txt file -> start the JAIG lifecycle **without sending it to the AI**

.java + selected code -> JAIG **refactoring mode**

JAIG Prompt Directives



All directives should be placed in the beginning of line.

#temperature: 0.0

#model: gpt-4 (you can find the available models in JAIG/JAIG.yaml)

#src – write to the **src/main/java** folder (more precisely, to the **srcFolder** folder configured in JAIG.yaml)

#test – write to the **src/test/java** folder (more precisely, to the **testFolder** folder configured in JAIG.yaml)

#save-to: path – write the AI's response to the specified file on the path path

#nomerge – prevent merging

#merge – allow merging (same as createMerge: yes in JAIG.yaml)

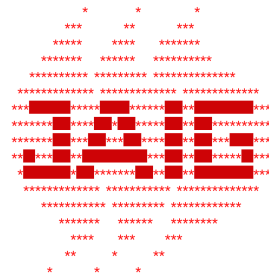
#patch – create a patch (same as createPatch: yes in JAIG.yaml)

#apply-merge – apply merge automatically

#norollback – prevent automatic rollback before re-running (if there is a .rollback file)

#merge-incomplete: <comma-separated list of classes> - merge existing code with GPT output(if response is incomplete)

All other lines starting with # are considered comments



Insert a folder in the middle of a prompt sequence

- Often, in the list of folders to generate, you need to **insert the folder in the middle** of the sequence
- If you apply JAIG to a folder like NN_something, such as 05_prompt, it will prompt you to enter a folder name
- When a name is entered, the folder will be named 06_folder_name, and all old folders 06_prompt, 07_prompt, etc., will receive an index one more.

1) click on the folder, press JAIG

- 01_rest_test
- 02_rest
- 03_controller
- 04_service
- 05_repository
- 06_unit_test

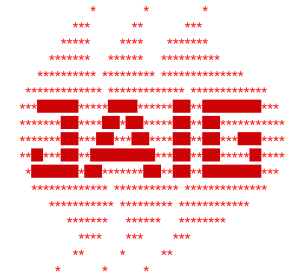


2) Enter the name of the folder: **dto**

3) A new folder **05_dto** inserted after 04_service:

- 01_rest_test
- 02_rest
- 03_controller
- 04_service
- 05_dto
- 06_repository
- 07_unit_test

Rollbacks



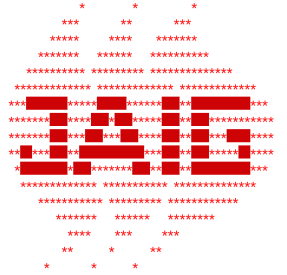
- When the code is generated, a **.rollback** file is created
- We can apply JAIG to **.rollback** file and all the changes will rollback **1 step back**.
- We have 2 possible .rollback files:
 - **<prompt>-parsed.rollback** allows to go **1 step back**
 - **<prompt>-full.rollback** allows to revert to the state before prompt was applied first time
- Rollbacks are **applied automatically** (if **applyRollback: yes** is set in **JAIG.yaml**) to be able to improve the request – this way we **start over every time**
- This means that **before the prompt is launched**, all **previous changes will be rolled back**, regardless of the changes made to the source code
- This is useful for **initial polishing of the prompt**, but is not suitable if you are going to change the source code (in this case, use \$patch or #merge)
- Rollbacks are **never applied automatically** if one of these directives used: **#patch**, **#merge**, **#merge-incomplete**, **#norollback**
 - **Patching** and **merging** need the folders like **backup** and **parsed-old not to be removed**, that's why it is incompatible with automatic rollbacks)

Here's an example of rollback file: **create-enum-parsed.rollback**: we restore the updated sources from the backup, cancelling all changes done by JAIG.

```
Restore src/main/java/test/Semaphore.java
from    JAIG/test-parse/create-enum-backup/test/Semaphore.java
Restore src/main/java/test/SemaphoreManager.java
from    JAIG/test-parse/create-enum-backup/test/SemaphoreManager.java
```

- If **applyRollback: no** is set in **JAIG.yaml**, you can add a **#rollback** directive to a prompt to apply the rollback automatically

Parsing <prompt>-response.txt



- After receiving the results of AI generation, a <prompt>-response.txt file is created with a response from the AI
- We can **apply JAIG to <prompt>-response.txt** instead of prompt
- In this case, the file content is parsed and patched, **without** having to **call the AI** again
- **This is useful for:**
 - **Demonstrations:** In this case, you can demonstrate how JAIG works (parsing, patching, etc.) without having to rely on the GPT generation (if you already have <prompt>-response.txt)
 - **Problems in the response parsing** (for example, the package before each class is not specified or incorrectly specified) which prevents response from parsing and we need to **make changes manually before parsing**
 - Parsing is possible ONLY if number of classes/interfaces (or other artifacts defined in javaFileNameRegexp) is the same as number of packages
 - Otherwise, we can change prompt and **ask to generate package** for every class OR change <prompt>-response.txt manually

If we are not satisfied with the results of parsing (for example, it wasn't able to find Java classes), we can:

- Roll back changes (apply JAIG to .rollback)
- Edit <prompt>-response.txt file
- Apply JAIG to <prompt>-response.txt