

AI assistants in Glamorous Toolkit
andrei.chis@feenk.com

Glamorous Toolkit

gt Page x

Page

"Generate the code to get the list of opened issues from the gtoolkit project in the feenkcom organization on github"

a GtLlmActionChat

Chat Instance Messages Write Tools Assistant formats Assistant act

You: Generate the code to get the list of opened issues from the gtoolkit project in the feenkcom organization on github

GPT 4.1: To fetch the list of open issues from the `gtoolkit` project under the `feenkcom` organization on GitHub in Pharo/Smalltalk, you typically use the `ZnClient` (Zinc HTTP Client) to perform a GET request on the GitHub REST API. Here is the code:

```
dict := ZnClient new
    url: 'https://api.github.com/repos/feenkcom/gtoolkit/issues?state=open';
    accept: ZnMimeType applicationJson;
    get.
issues := NeoJSONReader fromString: dict contents.
issues collect: [ :issue | issue at: 'title'].
```

This code retrieves all open issues and extracts their titles. You can further inspect each issue dictionary for more details as needed.

Text Code snippet Code Variables Diff Arguments JSON data

Chat Explain snippet Propose code Propose a view Propose an action Explain the object

Chat using the text field.

GPT 4.1

Glamorous Toolkit

gt AI assistants - presentation script

a GtGitHubIssues

GitHub Issues Raw Print Meta

Class

GtGitHubIssues

Superclass: Object Package: GtGitHub Tag:

Comment References References in snippets Advice definitions

Methods up to GtGitHubIssues Category All methods

Instance slots All methods read/write +

defaultOrganization llm-generated instance

defaultRepository llm-generated instance

fetchIssues llm-generated instance

gtIssuesListFor: aView

```
<gtView>
  ^ aView list
    title: 'GitHub Issues';
    items: [ issues ];
    itemText: [ :each | each at: 'title' ]
```

initialization Open chat

issuesTitles

a GtLlmActionChat

Chat Instance Messages Write Tools Assistant formats Assistant act

You: Convert this view method from a list into a columned list, and include a column showing the title and

GPT 4.1: To convert the current list view to a columned list with separate columns for title and author.

Text Code JSON data Changes

RBCompositeRefactoryChange

GtGitHubIssues>>gtIssuesListFor:

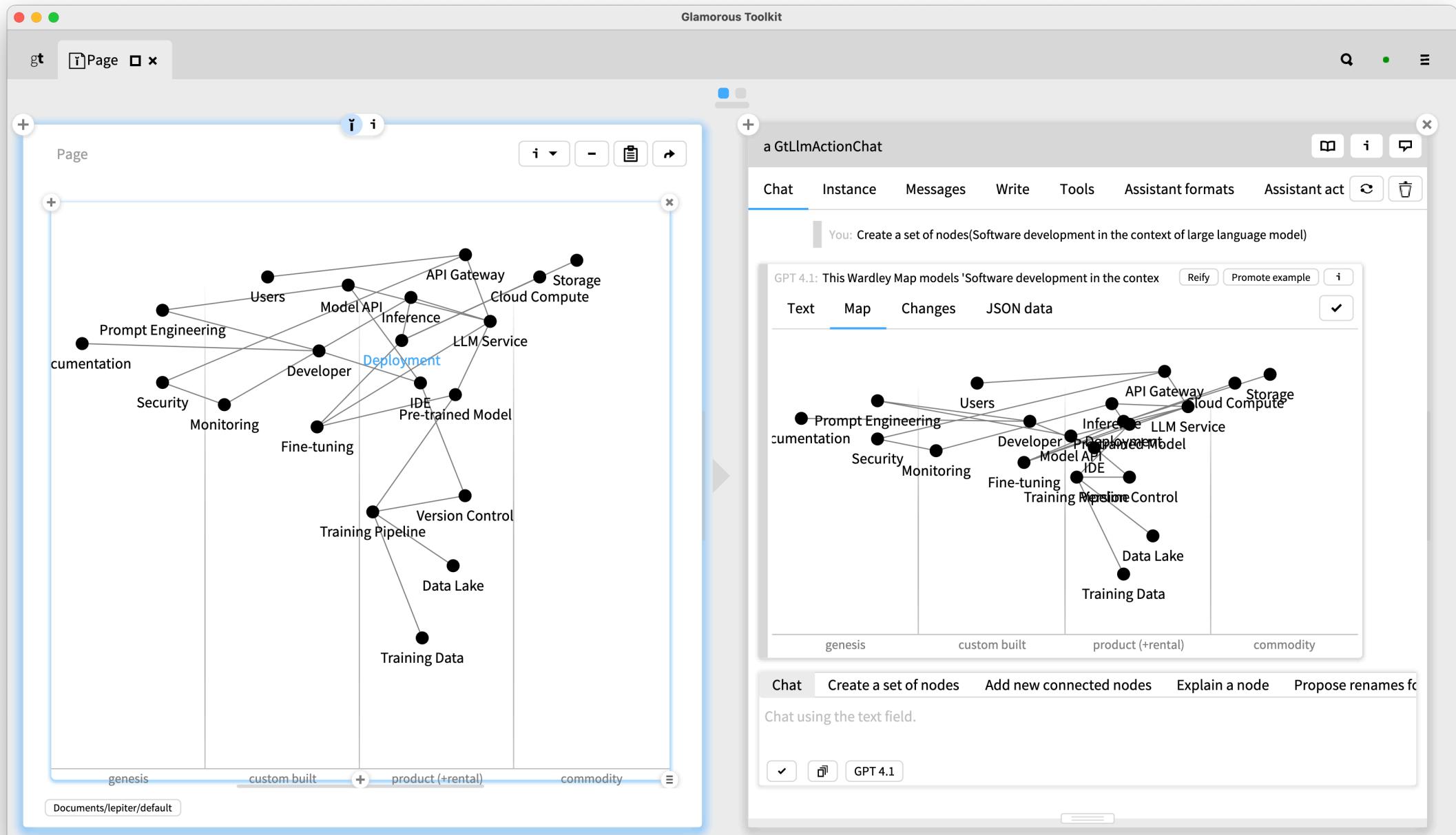
```
gtIssuesListFor: aView
  <gtView>
    ^ aView list
      title: 'GitHub Issues';
      items: [ issues ];
      itemText: [ :each | each at: 'title' ]
```

gtIssuesListFor: aView
 <gtView>
 ^ aView columnedList
 title: 'GitHub Issues';
 items: [issues];
 column: 'Title'
 text: [:each | each at: 'title'];
 column: 'Author'
 text: [:each | (each at: 'user') at: 'login']

Chat Rename method Review method Propose code Explain the object Explain the me

Chat using the text field.

GPT 4.1



Glamorous Toolkit

gt Building an ...ng blog posts x

Building an LLM assistant for editing blog posts

LLMs work interestingly well for shallow questions (I call shallow the questions whose answers can be evaluated against reality quickly). Moldable Development splits technical decision-making into small, shallow problems addressed by small, shallow tools. As such, it can be a fertile target for LLMs.

```
blogPostText := thisSnippet previousSibling ▶ text
```

And now, let's say we want to have a chat about it with a dedicated LLM:

```
GtOpenAIBlogPostAssistant ▶ new ▶ createChatOn ▶ blogPostText
```

Building a dedicated chat ... while chatting

The above assistant has been evolved from a generic assistant. We can refine a generic assistant to get to a similar result. This page demonstrates how to do that.

We start by instantiating a generic assistant with a description.

```
assistant := GtLlmAssistant ▶ new ▶ description: ▶ 'You are an assistant that is used to interactively work on blog posts.'
```

We want the assistant to work with a Post format, so we create it and give information on how to serialize and deserialize the blog post for the assistant.

Glamorous Toolkit Book

a GtLlmActionChat

You: Correct grammar

GPT 4.1: Correct grammar

Post Diff JSON data

There are at least two ways to use LLMs to support the understanding of software systems: (A) use LLMs to generate summaries (B) use LLMs to generate tools with which you generate summaries	There are at least two ways to use LLMs to support the understanding of software systems: (A) using LLMs to generate summaries (B) using LLMs to generate tools with which you generate summaries
The difference might appear subtle, but it's significant. A number of recent articles propose mostly (A) as a direction. These use LLMs as solution generators.	The difference might appear subtle, but it is significant. A number of recent articles propose mostly (A) as a direction, using LLMs as solution generators.
However, any concrete solutions entail biases. When we get the final answer, these biases are invisible. The solution might be correct, but you'd have no real way of knowing what tradeoffs it contains. The alternative is to regard current LLMs as generators of possibilities that should be evaluated against reality before being relied upon for decision-making in a technical space.	However, any concrete solution entails biases. When we get the final answer, these biases are invisible. The solution might be correct, but we have no real way of knowing what tradeoffs it contains. The alternative is to regard current LLMs as generators of possibilities that should be evaluated against reality before being relied upon for decision-making in a technical space.

Top Left: GitHub Issues View

The screenshot shows the GitHub Issues view within a Glimerous Toolkit window. It displays a list of issues from the "feenckon" organization. A snippet of code is shown to convert a list into a columned list:

```
gtIssuesListFor: <view>
  <view>
    title: 'GitHub Issues';
    items: [ issues ];
    itemText: [ each | each at: 'title' ]
  </view>
</view>
```

Top Right: Building an LLM Assistant

This window shows a Chat interface for building an LLM assistant. A snippet of code defines a blog post text:

```
blogPostText := thisSnippet previousSibling text;
```

The interface includes tabs for Chat, Review method, Rename method, Propose code, Explain the object, and Explain the code.

Bottom Left: Generating GitHub Issues Code

This window shows a Chat interface for generating code to fetch GitHub issues. A snippet of ZnClient code is provided:

```
dfc := ZnClient new.
dfc url: 'https://api.github.com/repos/feenckon/gitbook/issues?state=open';
dfc accept: 'application/json';
dfc get.
issues := JSONReader fromString: dfc contents.
issues collect: [ issue | issue at: 'title' ].
```

Bottom Right: Software Development Workflow

This window shows a Chat interface for creating a workflow diagram. The diagram illustrates a complex software development process involving multiple nodes like API Gateway, Storage, Cloud Compute, and various engineering and developer roles.

gtoolkit.com