Beginner.app will be a web-based SaaS application designed to help users explore and learn about various topics, from novice to expert. The platform uses an advanced spanning-tree language model that interacts with users through simple, engaging questions. By leveraging an advanced AI question generator and a unique level-based system, Beginner.app aims to revolutionize the learning experience.

Our tagline, “Imagine a tool like Google that asks **you, the user,** what you're interested in and has a genuine curiosity about helping you learn more. Beginner.app gets to know you from just a few simple prompts.” This idea has the potential to reshape the way we seek and process information. An AI that prompts learners worldwide.

Call to Action:
Are you ready to make a lasting impact in the EdTech industry? Think big, start small. Be part of a project that could revolutionize learning. Join the conversation tonight.

Expanded Market/Industry Analysis:

The global EdTech market is forecasted to grow at a CAGR of 16.3% from 2022 to 2028, reaching a value of $377.85 billion by 2028, according to Grand View Research. This growth is primarily driven by the increasing use of eLearning tools, a shift towards remote learning, and the ongoing integration of AI in the education sector.

Beginner.app positions itself within this burgeoning market by providing an AI-driven, personalized learning experience. Its unique spanning-tree language model and level-based system aim to engage users at their current level of understanding, thereby fostering an environment of progressive learning.

The target audience for Beginner.app is diverse, with a significant focus on lifelong learners - a demographic that spans from students to retirees. According to a report by Pew Research Center, 73% of American adults consider themselves lifelong learners. These figures are expected to increase worldwide, particularly in regions with rising internet penetration rates and smartphone usage. This trend suggests a broad and expanding market for Beginner.app's services.

In the professional sphere, continuous learning and skill development are becoming increasingly necessary. The World Economic Forum estimates that 54% of all employees will require significant re- and up-skilling by 2022. This need for ongoing professional development creates a potential user base for Beginner.app, particularly among those seeking accessible, self-paced learning solutions.

Furthermore, the casual learning market, which includes hobbyists and those learning for personal interest or satisfaction, also presents a significant opportunity. A report by Ofcom indicates that during the pandemic, people turned to online learning platforms to cultivate new

hobbies and learn new skills. While the pandemic may have accelerated this trend, the desire to learn and engage in personal development remains constant.

On the competitive front, the AI in education market is fragmented with several key players such as Coursera, Duolingo, and Quizlet. However, Beginner.app's unique approach of an AI-driven personalized learning journey, utilizing an engaging question format, sets it apart. According to a report by eLearning Industry, personalized learning increases student engagement and retention rates, indicating a high potential for success for Beginner.app.

However, potential challenges for Beginner.app might include user data privacy and security concerns, the digital divide in underdeveloped regions, and the need for constant technological upgrades to stay competitive. These aspects must be carefully managed to ensure sustained growth and success in the market.

In conclusion, the rapidly expanding EdTech market, combined with the rising demand for personalized, lifelong learning, presents a promising opportunity for Beginner.app. Its unique AI-driven approach and broad target audience set it apart in a fragmented market, positioning it as a potential frontrunner in the AI-driven learning space.

Plugin authentication

Plugins offer numerous authentication schemas to accommodate various use cases. To specify the authentication schema for your plugin, use the manifest file. Our plugin domain policy outlines our strategy for addressing domain security issues. For examples of available authentication options, refer to the examples section, which showcases all the different choices.

The `ai-plugin.json` file requires an `auth` schema to be set. Even if you elect to use no authentication, it is still required to specify `"auth": { "type": "none" }`.

We support only localhost development without authentication; if you want to use service, user, or OAuth authentication, you need to set up a remote server.

No authentication

We support no-auth flow for applications that do not require authentication, where a user is able to send requests directly to your API without any restrictions. This is particularly useful if you have an open API that you want to make available to everyone, as it allows traffic from sources other than just OpenAI plugin requests.

```
"auth": {
  "type": "none"
},
```

Service level

If you want to specifically enable OpenAI plugins to work with your API, you can provide a client secret during the plugin installation flow. This means that all traffic from OpenAI plugins will be authenticated but not on a user level. This flow benefits from a simple end user experience but less control from an API perspective.

- To start, select "Develop your own plugin" in the ChatGPT plugin store, and enter the domain where your plugin is hosted.
- In `ai-plugin.json`, set `auth.type` to `"service_http"` as is shown in our service level auth example.
- You will be prompted for your service access token, which is a string specified in your code.
    - We securely store an encrypted copy of your service access token to enable plugin installation without additional authentication.

    - The service access token is sent in the `Authorization` header for plugin requests.
- Once you add your service access token into the ChatGPT UI, you will be presented with a verification token.
- Add the verification token to your `ai-plugin.json` file under the auth section as shown below.

```
"auth": {
  "type": "service_http",
  "authorization_type": "bearer",
  "verification_tokens": {
    "openai": "Replace_this_string_with_the_verification_token_generated_in_the_ChatGPT_UI"
  }
},
```

The verification tokens are designed to support multiple applications. You can simply add the additional applications you want your plugin to support:

```
"verification_tokens": {
    "openai": "Replace_this_string_with_the_verification_token_generated_in_the_ChatGPT_UI",
    "other_service": "abc123"
  }
```

User level

Due to current UI limitations, we are not allowing plugins with User authentication into the plugin store. We expect this to change in the near future.

Just like how a user might already be using your API, we allow user level authentication through enabling end users to copy and paste their secret API key into the ChatGPT UI during plugin install. While we encrypt the secret key when we store it in our database, we do not recommend this approach given the poor user experience.

- To start, a user pastes in their access token when installing the plugin
- We store an encrypted version of the token
- We then pass it in the Authorization header when making requests to the plugin ("Authorization": "[Bearer/Basic] [user's token]")

```
"auth": {
  "type": "user_http",
  "authorization_type": "bearer",
},
```

OAuth

The plugin protocol is compatible with OAuth. A simple example of the OAuth flow we are expecting should look something like the following:

- To start, select "Develop your own plugin" in the ChatGPT plugin store, and enter the domain where your plugin is hosted (cannot be localhost).
- In `ai-plugin.json`, set `auth.type` to `"oauth"` as is shown in our OAuth example.
- Then, you will be prompted to enter the OAuth client ID and client secret.
  - The client ID and secret can be simple text strings but should follow OAuth best practices.
  - We store an encrypted version of the client secret, while the client ID is available to end users.
- Once you add your client ID and client secret into the ChatGPT UI, you will be presented with a verification token.
- Add the verification token to your `ai-plugin.json` file under the auth section as shown below.
- OAuth requests will include the following information: `request={'grant_type': 'authorization_code', 'client_id': 'id_set_by_developer', 'client_secret': 'secret_set_by_developer', 'code': 'abc123', 'redirect_uri': 'https://chat.openai.com/aip/plugin-some_plugin_id/oauth/callback'}`
- In order for someone to use a plugin with OAuth, they will need to install the plugin and then be presented with a "Sign in with" button in the ChatGPT UI.
- The `authorization_url` endpoint should return a response that looks like: `{ "access_token": "example_token", "token_type": "bearer", "refresh_token": "example_token", "expires_in": 59, }`
- During the user sign in process, ChatGPT makes a request to your `authorization_url` using the specified `authorization_content_type`, we expect to get back an access token and optionally a refresh token which we use to periodically fetch a new access token.
- Each time a user makes a request to the plugin, the user's token will be passed in the Authorization header: ("Authorization": "[Bearer/Basic][user's token]").

Below is an example of what the OAuth configuration inside of the `ai-plugin.json` file might look like:

```
10
"auth": {

"type": "oauth",
"client_url": "https://example.com/authorize",
"scope": "",
"authorization_url": "https://example.com/auth/",
"authorization_content_type": "application/json",
"verification_tokens": {


"openai": "Replace_this_string_with_the_verification_token_generated_in_the_ChatGPT_UI"

 }

},
```

To better understand the URL structure for OAuth, here is a short description of the fields:

- When you set up your plugin with ChatGPT, you will be asked to provide your OAuth `client_id` and `client_secret`.
- When a user logs into the plugin, ChatGPT will direct the user's browser to `"[client_url]?response_type=code&client_id=[client_id]&scope=[scope]&redirect_uri=https%3A%2F%2Fchat.openai.com%2Faip%2F[plugin_id]%2Foauth%2Fcallback"`
- The `plugin_id` is passed via the request made to your OAuth endpoint (note that it is not visible in the ChatGPT UI today but may be in the future). You can inspect the request there to see the `plugin_id`.
- After your plugin redirects back to the given `redirect_uri`, ChatGPT will complete the OAuth flow by making a POST request to the `authorization_url` with content type `authorization_content_type` and parameters `{ "grant_type": "authorization_code", "client_id": [client_id], "client_secret":`

[client_secret], “code”: [the code that was returned with the redirect], “redirect_uri”: [the same redirect uri as before] }.

Getting started

Creating a plugin takes 3 steps:

1. Build an API
2. Document the API in the OpenAPI yaml or JSON format
3. Create a JSON manifest file that will define relevant metadata for the plugin

The focus of the rest of this section will be creating a todo list plugin by defining the OpenAPI specification along with the manifest file.

Explore example plugins
Explore example plugins covering multiple use cases and authentication methods.

Plugin manifest

Every plugin requires a `ai-plugin.json` file, which needs to be hosted on the API's domain. For example, a company called `example.com` would make the plugin JSON file accessible via an https://example.com domain since that is where their API is hosted. When you install the plugin via the ChatGPT UI, on the backend we look for a file located at `/.well-known/ai-plugin.json`. The `/.well-known` folder is required and must exist on your domain in order for ChatGPT to connect with your plugin. If there is no file found, the plugin cannot be installed. For local development, you can use HTTP but if you are pointing to a remote server, HTTPS is required.

The minimal definition of the required `ai-plugin.json` file will look like the following

```
{
  "schema_version": "v1",
  "name_for_human": "TODO Plugin",
  "name_for_model": "todo",
  "description_for_human": "Plugin for managing a TODO list. You can add, remove and view your TODOs.",
  "description_for_model": "Plugin for managing a TODO list. You can add, remove and view your TODOs.",
  "auth": {
    "type": "none"
  },
  "api": {
    "type": "openapi",
    "url": "http://localhost:3333/openapi.yaml",
    "is_user_authenticated": false
  },
  "logo_url": "http://localhost:3333/logo.png",
  "contact_email": "support@example.com",
  "legal_info_url": "http://www.example.com/legal"
}
```

If you want to see all of the possible options for the plugin file, you can refer to the definition below. When naming your plugin, please keep in mind our brand guidelines, plugins that fail to adhere to these guidelines will not be approved for the plugin store.

Field	Type	Description / Options	Required
`schema_version`	String	Manifest schema version	
`name_for_model`	String	Name the model will use to target the plugin (no spaces allowed, only letters and numbers). 50 character max.	
`name_for_human`	String	Human-readable name, such as the full company name. 20 character max.	
`description_for_model`	String	Description better tailored to the model, such as token context length considerations or keyword usage for improved plugin prompting. 8,000 character max.	
`description_for_human`	String	Human-readable description of the plugin. 100 character max.	
`auth`	ManifestAuth	Authentication schema	
`api`	Object	API specification	
`logo_url`	String	URL used to fetch the logo. Suggested size: 512 x 512. Transparent backgrounds are supported.	
`contact_email`	String	Email contact for safety/moderation, support, and deactivation	
`legal_info_url`	String	Redirect URL for users to view plugin information	
`HttpAuthorizationType`	HttpAuthorizationType	"bearer" or "basic"	
`ManifestAuthType`	ManifestAuthType	"none", "user_http", "service_http", or "oauth"	
`interface` BaseManifestAuth	BaseManifestAuth	type: ManifestAuthType; instructions: string;	
`ManifestNoAuth`	ManifestNoAuth	No authentication required: BaseManifestAuth & { type: 'none', }	
`ManifestAuth`	ManifestAuth	ManifestNoAuth, ManifestServiceHttpAuth, ManifestUserHttpAuth, ManifestOAuthAuth	

The following are examples with different authentication methods:

```
# App-level API keys
type ManifestServiceHttpAuth  = BaseManifestAuth & {
  type: 'service_http';
  authorization_type: HttpAuthorizationType;
  verification_tokens: {
    [service: string]?: string;
  };
}

# User-level HTTP authentication
type ManifestUserHttpAuth  = BaseManifestAuth & {
  type: 'user_http';
  authorization_type: HttpAuthorizationType;
}

type ManifestOAuthAuth  = BaseManifestAuth & {
  type: 'oauth';

  # OAuth URL where a user is directed to for the OAuth authentication flow to begin.
  client_url: string;

  # OAuth scopes required to accomplish operations on the user's behalf.
  scope: string;

  # Endpoint used to exchange OAuth code with access token.
  authorization_url: string;

  # When exchanging OAuth code with access token, the expected header 'content-type'. For example: 'content-type: application/json'
  authorization_content_type: string;

  # When registering the OAuth client ID and secrets, the plugin service will surface a unique token.
  verification_tokens: {
    [service: string]?: string;
  };
}
```

There are limits to the length of certain fields in the manifest file mentioned above which are subject to change. We also impose a 100,000 character maximum for the API response body which may also change over time.

In general, the best practice is to keep the description and responses as concise as possible because the models have limited context windows.

OpenAPI definition

The next step is to build the OpenAPI specification to document the API. The model in ChatGPT does not know anything about your API other than what is defined in the OpenAPI specification and manifest file. This means that if you have an extensive API, you need not expose all functionality to the model and can choose specific endpoints. For example, if you have a social media API, you might want to have the model access content from the site through a GET request but prevent the model from being able to comment on users posts in order to reduce the chance of spam.

The OpenAPI specification is the wrapper that sits on top of your API. A basic OpenAPI specification will look like the following:

```
openapi: 3.0.1
info:
  title: TODO Plugin
  description: A plugin that allows the user to create and manage a TODO list using ChatGPT.
  version: 'v1'
servers:
  - url: http://localhost:3333
paths:
  /todos:
    get:
      operationId: getTodos
      summary: Get the list of todos
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/getTodosResponse'
components:
  schemas:
    getTodosResponse:
      type: object
      properties:
        todos:
          type: array
          items:
            type: string
          description: The list of todos.
```

We start by defining the specification version, the title, description, and version number. When a query is run in

ChatGPT, it will look at the description that is defined in the info section to determine if the plugin is relevant for the user query. You can read more about prompting in the writing descriptions section.

Keep in mind the following limits in your OpenAPI specification, which are subject to change:

- 200 characters max for each API endpoint description/summary field in API specification
- 200 characters max for each API param description field in API specification

Since we are running this example locally, we want to set the server to point to your localhost URL. The rest of the OpenAPI specification follows the traditional OpenAPI format, you can learn more about OpenAPI formatting through various online resources. There are also many tools that auto generate OpenAPI specifications based on your underlying API code.

Running a plugin

Once you have created an API, manifest file, and OpenAPI specification for your API, you are now ready to connect the plugin via the ChatGPT UI. There are two different places your plugin might be running, either locally in a development environment or on a remote server.

If you have a local version of your API running, you can point the plugin interface to your localhost server. To connect the plugin with ChatGPT, navigate to the plugin store and select "Develop your own plugin". Enter your localhost and port number (e.g `localhost:3333`). Note that only auth type `none` is currently supported for localhost development.

If the plugin is running on a remote server, you will need to first select "Develop your own plugin" to set it up and then "Install an unverified plugin" to install it for yourself. You can simply add the plugin manifest file to the `yourdomain.com/.well-known/` path and start testing your API. However, for subsequent changes to your manifest file, you will have to deploy the new changes to your public site which might take a long time. In that case, we suggest setting up a local server to act as a proxy for your API. This allows you to quickly prototype changes to your OpenAPI spec and manifest file.

Setup a local proxy of your public API

The following Python code is an example of how you can set up a simple proxy of your public facing API.

```
import requests
import os

import yaml
from flask import Flask, jsonify, Response, request, send_from_directory
from flask_cors import CORS

app = Flask(__name__)

PORT = 3333

# Note: Setting CORS to allow chat.openapi.com is required for ChatGPT to access your plugin
CORS(app, origins=[f"http://localhost:{PORT}", "https://chat.openai.com"])

api_url = 'https://example.com'


@app.route('/.well-known/ai-plugin.json')
def serve_manifest():
    return send_from_directory(os.path.dirname(__file__), 'ai-plugin.json')


@app.route('/openapi.yaml')
def serve_openapi_yaml():
    with open(os.path.join(os.path.dirname(__file__), 'openapi.yaml'), 'r') as f:
        yaml_data = f.read()
    yaml_data = yaml.load(yaml_data, Loader=yaml.FullLoader)
    return jsonify(yaml_data)


@app.route('/openapi.json')
def serve_openapi_json():
    return send_from_directory(os.path.dirname(__file__), 'openapi.json')


@app.route('/<path:path>', methods=['GET', 'POST'])
def wrapper(path):

    headers = {
    'Content-Type': 'application/json',
    }

    url = f'{api_url}/{path}'
    print(f'Forwarding call: {request.method} {path} -> {url}')

    if request.method == 'GET':
        response = requests.get(url, headers=headers, params=request.args)
    elif request.method == 'POST':
        print(request.headers)
        response = requests.post(url, headers=headers, params=request.args, json=request.json)
    else:
        raise NotImplementedError(f'Method {request.method} not implemented in wrapper for {path=}')
    return response.content


if __name__ == '__main__':
    app.run(port=PORT)
```

Writing descriptions

When a user makes a query that might be a potential request that goes to a plugin, the model looks through the descriptions of the endpoints in the OpenAPI specification along with the `description_for_model` in the manifest file. Just like with prompting other language models, you will want to test out multiple prompts and descriptions to see what works best.

The OpenAPI spec itself is a great place to give the model information about the diverse details of your API – what functions are available, with what parameters, etc. Besides using expressive, informative names for each field, the spec can also contain "description" fields for every attribute. These can be used to provide natural language descriptions of what a function does or what information a query field expects, for example. The model will be able to see these, and they will guide it in using the API. If a field is restricted to only certain values, you can also provide an "enum" with descriptive category names.

The `description_for_model` attribute gives you the freedom to instruct the model on how to use your plugin generally. Overall, the language model behind ChatGPT is highly capable of understanding natural language and following instructions. Therefore, this is a good place to put in general instructions on what your plugin does and how the model should use it properly. Use natural language, preferably in a concise yet descriptive and objective tone. You can look at some of the examples to have an idea of what this should look like. We suggest starting the `description_for_model` with "Plugin for …" and then enumerating all of the functionality that your API provides.

Best practices

Here are some best practices to follow when writing your `description_for_model` and descriptions in your OpenAPI specification, as well as when designing your API responses:

1. Your descriptions should not attempt to control the mood, personality, or exact responses of ChatGPT. ChatGPT is designed to write appropriate responses to plugins.

   Bad example:

   > When the user asks to see their todo list, always respond with "I was able to find your todo list! You have [x] todos: [list the todos here]. I can add more todos if you'd like!"

   Good example:

   > [no instructions needed for this]

2. Your descriptions should not encourage ChatGPT to use the plugin when the user hasn't asked for your plugin's particular category of service.

   Bad example:

   > Whenever the user mentions any type of task or plan, ask if they would like to use the TODOs plugin to add something to their todo list.

Good example:

> The TODO list can add, remove and view the user's TODOs.

3. Your descriptions should not prescribe specific triggers for ChatGPT to use the plugin. ChatGPT is designed to use your plugin automatically when appropriate.

   Bad example:

   > When the user mentions a task, respond with "Would you like me to add this to your TODO list? Say 'yes' to continue."

   Good example:

   > [no instructions needed for this]

4. Plugin API responses should return raw data instead of natural language responses unless it's necessary. ChatGPT will provide its own natural language response using the returned data.

   Bad example:

   > I was able to find your todo list! You have 2 todos: get groceries and walk the dog. I can add more todos if you'd like!

   Good example:

   > { "todos": ["get groceries", "walk the dog"] }

Debugging

By default, the chat will not show plugin calls and other information that is not surfaced to the user. In order to get a more complete picture of how the model is interacting with your plugin, you can see the request and response by clicking the down arrow on the plugin name after interacting with the plugin.

A model call to the plugin will usually consist of a message from the model containing JSON-like parameters which are sent to the plugin, followed by a response from the plugin, and finally a message from the model utilizing the information returned by the plugin.

If you are developing a localhost plugin, you can also open the developer console by going to "Settings" and toggling "Open plugin devtools". From there, you can see more verbose logs and "refresh plugin" which re-fetches the Plugin and OpenAPI specification.