Proposed Prompting Methodology

For your projects, you will need to generate frontend and backend code using generative (LLMs) models. The success and effort of this will depend on the prompting methodology you will follow and the models you will choose to use.

For this reason, we suggest the following guidelines and provide prompting templates one can utilize as a starting point.

Prompting Methodology:

Context refers to the knowledge about a conversation a model has during that conversation. Context is limited in all models, but some have greater limits than the others. For maximum efficiency, good practices dictate not to provide multiple times the same context (e.g. files) that you have already given to the LLMs, and try to keep context in mind, especially when generating large coding projects.

Since the frontend is usually the calling entity and the backend is the receiving/responding entity, it is a good idea to first start generating the backend.

Generating the Backend

One can start generating the backend using the following template prompt:

```
---------------------------------------------------------------------

Generate Production-Ready Node.js/Express REST API

**Name:** {{PROJECT_NAME}}

**Description:** {{PROJECT_DESCRIPTION}}
```

Input Sources (Priority Order)

OpenAPI Spec (PRIMARY) - All endpoints, schemas, validation

User Stories - Feature context & flows

Requirements - Business rules & constraints

TARGET STRUCTURE

package.json - Dependencies

.env.example - environmental variables

.gitignore - Git ignore rules

models/.js* - Mongoose models for each schema

routes/.js* - Express routers for each resource

controllers/.js* - Controller functions

services/.js* - Business logic layer

middleware/ - auth.js, validation.js, error Handler.js, logger.js

config/ - database.js, constants.js, …

utils/ - responses.js, validators.js, helpers.js

server.js & app.js - Entry points

README.md - Documentation

CODE STYLE REQUIREMENTS

Async Pattern: async/await

Error Handling: try-catch with centralized error middleware

Response Pattern: {success, data, error, message}

Naming: camelCase for variables/functions, PascalCase for models

Imports: use ES6 modules

SOURCE ARTIFACTS

1. OPENAPI/SWAGGER SPECIFICATION

{{SWAGGER_JSON}}

2. USER STORIES

{{USER_STORIES}}

3. REQUIREMENTS

{{REQUIREMENTS_LIST}}

CRITICAL REQUIREMENTS

COMPLETE, RUNNABLE code (no placeholders)

Every controller function has try-catch

All async functions use async/await

All functions have JSDoc comments

Use ES6+ features

All models use Mongoose Schema properly

All routes mounted in routes/index.js

Error handler differentiates error types

Use basic authentication

There must be at least 10 (ten) available routes/API calls that an external user can access and receive a response from.

There must be at least one GET route, one POST route, one PUT route, and one DELETE route – ideally referring to the same resource, so that its full functionality can be tested.

The system must include at least 3 (three) different entities (e.g., users, books, libraries) that interact with each other across various routes (e.g., one route records the borrowing of a book by a user, another checks if a book is available in a library, etc.).

For the purposes of the routes – and since some data will also need to be displayed on the frontend – mock data should be used, hardcoded within the backend code, so that they are immediately available every time the server runs.

```
Optionally, you may connect your backend to a MongoDB Atlas database if
you want the data to be stored persistently.

The mock data can be simple example objects (e.g., lists of users, books,
or libraries), as long as they serve the needs of the frontend screens and
demonstrate the core functionality of the application. The data does not
need to be realistic or large-scale.

Make it so that if mongo URI is not provided, it falls back to just mock
data.

Generate the complete backend.
```

Double brackets {{}} are placeholders where you can insert the respective information, or instead, you can choose to attach that information to the conversation.

Depending on the utilized models, this will require some back and forth to achieve a fully functional backend. It is a good approach to provide the error messages to the LLMs, on each step, as well as screenshots when needed, especially for frontend errors.

After generating the backend, testing its functionality using http requests with tools like (curl (command line) or Postman) proceed to generating the frontend.

Generating the Frontend

After building the backend, one has to proceed with generating the frontend. It is suggested that the same conversation is reused in order to maintain context from previous interactions. This way the frontend can be fine-tuned on the backend's specifications, offering greater chances of error-free generation. However, if this is not possible, one can ask the LLM for a recap of the backend-generation conversation, to transfer some of the previous knowledge to the new, frontend-generation conversation.

The following prompt can be utilized for generating the base frontend:

```
# Generate Production-Ready React Frontend

**Name:** {{PROJECT_NAME}}

**Description:** {{PROJECT_DESCRIPTION}}


## Input Sources (Priority Order)
```

1. **Mockups/Wireframes** (PRIMARY) - Visual structure & layout

2. **OpenAPI Spec** - API endpoints & data models

3. **User Stories** - User flows & interactions

4. **Activity Diagrams** - User journeys

SOURCE ARTIFACTS

1. MOCKUPS (for page structure and navigation):

{{MOCKUPS}}

2. OPENAPI/SWAGGER SPECIFICATION (PRIMARY SOURCE for API/backend):

{{OPENAPI_SPEC}}

3. USER STORIES (for UI features and user flows):

{{USER_STORIES}}

4. ACTIVITY DIAGRAMS (for UI features and user flows)

{{ACTIVITY DIAGRAMS}}

##FRONTEND STRUCTURE

package.json - Dependencies

.env.example - Environment variables

.gitignore - Git ignore rules

README.md - Documentation

public/*.* - Static assets & index.html

src/index.js - App entry point

src/App.js - Root component

src/index.css - Global styles

src/api/*.js - API client & methods for each resource

src/components/*.jsx - Reusable UI components

src/pages/*.jsx - Page components

src/hooks/*.js - Custom React hooks

src/context/*.jsx - React Context providers

src/router/*.jsx - Route definitions & protected routes

src/utils/*.js - Validators, formatters, constants, helpers

src/styles/*.css - Global styles & CSS variables

Output Requirements

Generate complete React app with:

- Components from mockups
- Pages/routes from user flows
- API service layer (axios)
- State management
- Form validation
- Responsive design
- package.json with all deps
- Fetching data using backend endpoints (described in OpenAPI Spec)
- Use basic authentication
- Use create-react-app

Frontend Requirements:

The frontend must consist of at least 4 (four) screens.

It is recommended that these screens correspond to the provided mockups, but you may also include additional screens if you consider them necessary.

The frontend must make requests to at least 5 (five) of the available backend endpoints.

Generate complete frontend.

Again, double brackets {{}} are placeholders where you can insert the respective information, or instead, you can choose to attach that information to the conversation.

Back and forth may be necessary in order to generate a fully functional frontend. Make sure you test both applications and their communication, to ensure the integrate and exchange information smoothly.

Suggested Models:

It is always suggested to utilize the latest available models, rather than older, deprecated ones. All models, including their free-versions will work fine for code generation. However, some models excel over others. As students, you are entitled to the following premium models at the date of writing these guidelines:

- Github Copilot Pro which integrates smoothly with VS Code for development.
- Google Gemini Pro for students – Offer expires on December 9, 2025

Admittedly, Github Copilot Pro + VS Code integration seems to outperform most coding generation methods we have tried. Furthermore, some models such as the latest Claude 4.5, Gemini Pro and ChatGPT 5 seem to outperform the others in coding generation.