# Prompt Parsing Process Design Approaches

For parsing the `IPrompt_v3_for_promptsFactory[]` data structure, we can consider the following approaches to transform it into precise prompts for large language models:

## 1. Template-Based Concatenation

Parse different prompt types using predefined templates:

```
function parsePrompts(prompts: IPrompt_v3_for_promptsFactory[]): string
  const selectedPrompts = prompts.filter(p => p.status === 'selected');
  let finalPrompt = '';

  // Group by type and concatenate in specific order
  const context = selectedPrompts.filter(p => p.type === EPrompt_v3_for_
  if (context.length) finalPrompt += `## Background\n${context.map(p =>

  const role = selectedPrompts.filter(p => p.type === EPrompt_v3_for_pro
  if (role.length) finalPrompt += `## Your Role\n${role.map(p => p.title

  // Process other types similarly...

  return finalPrompt;
}
```

## 2. Structured Assembly Method

Organize different prompt types into a structured JSON object, then convert to text based on predefined patterns:

```
function parsePrompts(prompts: IPrompt_v3_for_promptsFactory[]): string
  const selectedPrompts = prompts.filter(p => p.status === 'selected');
  const promptStructure = {
    context: selectedPrompts.filter(p => p.type === EPrompt_v3_for_promp
```

```
    role: selectedPrompts.filter(p => p.type === EPrompt_v3_for_promptsF
    // Other types...
  };

  return convertStructureToText(promptStructure);
}
```

## 3. Priority-Based Sorting

Sort prompt elements based on importance and logical order, then concatenate:

```
function parsePrompts(prompts: IPrompt_v3_for_promptsFactory[]): string
  const selectedPrompts = prompts.filter(p => p.status === 'selected');

  // Define type priorities
  const typePriority = {
    [EPrompt_v3_for_promptsFactory_type.CONTEXT]: 1,
    [EPrompt_v3_for_promptsFactory_type.ROLE]: 2,
    [EPrompt_v3_for_promptsFactory_type.OBJECTIVE]: 3,
    // Other type priorities...
  };

  // Sort by priority
  selectedPrompts.sort((a, b) => typePriority[a.type] - typePriority[b.t

  // Concatenate final prompt
  return selectedPrompts.map(p => `[${p.type}]: ${p.title}`).join('\n\n'
}
```

## 4. Conditional Dynamic Assembly

Adjust assembly strategy based on existing prompt types:

```
function parsePrompts(prompts: IPrompt_v3_for_promptsFactory[]): string
  const selectedPrompts = prompts.filter(p => p.status === 'selected');
```

```
  let finalPrompt = '';

  // Check for specific types
  const hasRole = selectedPrompts.some(p => p.type === EPrompt_v3_for_pr
  const hasObjective = selectedPrompts.some(p => p.type === EPrompt_v3_f

  // Dynamically adjust template based on existing types
  if (hasRole && hasObjective) {
    // Role-oriented task template
    finalPrompt = constructRoleBasedPrompt(selectedPrompts);
  } else if (hasObjective) {
    // Objective-oriented task template
    finalPrompt = constructObjectiveBasedPrompt(selectedPrompts);
  } else {
    // Default template
    finalPrompt = constructDefaultPrompt(selectedPrompts);
  }

  return finalPrompt;
}
```

## 5. Tag-Based Filtering Enhancement

Utilize tags property for more granular filtering and enhancement:

```
function parsePrompts(prompts: IPrompt_v3_for_promptsFactory[]): string
  const selectedPrompts = prompts.filter(p => p.status === 'selected');

  // Group by tags
  const promptsByTag = {};
  selectedPrompts.forEach(p => {
    if (p.tags) {
      p.tags.forEach(tag => {
        if (!promptsByTag[tag]) promptsByTag[tag] = [];
        promptsByTag[tag].push(p);
      });
    }
```

```
  });

  // Build specific prompt sections based on tag combinations
  // ...

  return finalPrompt;
}
```

## 6. History-Aware Recursive Parsing

Process prompts with history records by recursively parsing historical content:

```
function parsePrompts(prompts: IPrompt_v3_for_promptsFactory[], isRecur
  const selectedPrompts = prompts.filter(p => p.status === 'selected');
  let finalPrompt = '';

  for (const prompt of selectedPrompts) {
    finalPrompt += `[${prompt.type}]: ${prompt.title}\n`;

    // Recursively process history
    if (prompt.history && prompt.history.length > 0) {
      finalPrompt += `\nHistory:\n${parsePrompts(prompt.history, true)}\
    }
  }

  return finalPrompt;
}
```

## 7. Model-Specific Optimization

Customize prompt format based on target model characteristics:

```
function parsePrompts(prompts: IPrompt_v3_for_promptsFactory[], modelTy
  const selectedPrompts = prompts.filter(p => p.status === 'selected');
```

```
  switch (modelType) {
    case 'gpt4':
      return formatForGPT4(selectedPrompts);
    case 'claude':
      return formatForClaude(selectedPrompts);
    case 'llama':
      return formatForLlama(selectedPrompts);
    default:
      return formatDefault(selectedPrompts);
  }
}
```

## 8. Interactive Construction Method

Allow users to interactively adjust prompts during the construction process:

```
async function buildPromptInteractively(prompts: IPrompt_v3_for_prompts
  const selectedPrompts = prompts.filter(p => p.status === 'selected');
  let finalPrompt = '';

  // Show preliminary constructed prompt
  finalPrompt = basicParse(selectedPrompts);

  // Allow user adjustments
  const userFeedback = await getUserFeedback(finalPrompt);

  // Adjust based on feedback
  return adjustPromptBasedOnFeedback(finalPrompt, userFeedback);
}
```

These methods can be used individually or in combination to create the most suitable prompt parsing approach for specific use cases.

---

<small>Powered by GPT AI Flow</small>