INTRODUCTION

Systematic LLM experimentation at scale presents a significant engineering challenges. The goal is often to compare numerous models across various configurations (different prompts, parameters, datasets) to find the optimal setup for a given task. This immediately leads to a combinatorial explosion.

For instance, the foundational work leading to Kernel 29 – the dxgpt-testing-main project –, aimed to assess how effectively different LLMs could propose differential diagnoses based purely on simulated plain-text non-standarized clinical cases. The goal was to test various LLMs and prompts to see how well they could produce a ranked list of 5 potential diagnoses, supported by reasoning that compared the patient's symptoms to those of the suggested conditions(a differential diagnosis).

This required extensive testing across different setups to find the most effective combinations. Specifically:

- 18 distinct model configurations (Table 1) accessed via multiple APIs.
- 5 different prompting strategies (Table 2) tailored for diagnosis.
- 5 clinical case datasets (Table 3).

From a technical standpoint, this setup translates to managing 18 models × 5 prompts × 5 datasets = potential 450 unique experimental runs, even before considering variations in parameters (like temperature) or incorporating evaluation steps (e.g., using LLM-as-judges).

Table 1: Model Configurations

Alias	Source API / Service	Base Model
c3opus	Anthropic API	claude-3-opus-20240229
c35sonnet	Anthropic API	claude-3-5-sonnet-20240620
c3sonnet	AWS Bedrock	anthropic.claude-3-sonnet-v1:0
mistralmoe	AWS Bedrock	mistral.mixtral-8x7b-instruct-v0:1
mistral7b	AWS Bedrock	mistral.mistral-7b-instruct-v0:2
llama2_7b	Azure ML Endpoint	Llama-2-7b-chat-dxgpt
llama3_8b	Azure ML Endpoint	llama-3-8b-chat-dxgpt
llama3_70b	Azure ML Endpoint	llama-3-70b-chat-dxgpt
cohere_cplus	Azure ML Endpoint	Cohere-command-r-plus-dxgpt
geminipro	GCP Vertex AI	gemini-1.5-pro-preview-0409
gpt4_0613azure	Azure OpenAI Service	gpt-4 (0613 deployment)
gpt4turboazure	Azure OpenAI Service	gpt-4-turbo (1106 deployment)
gpt4turbo1106	OpenAI API	gpt-4-1106-preview
gpt4turbo0409	OpenAI API	gpt-4-turbo-2024-04-09
gpt4o	OpenAI API	gpt-4o
o1_mini	OpenAI API	o1-mini
o1_preview	OpenAI API	o1-preview
mistralmoebig	Mistral API	open-mixtral-8x22b

Table 2: Prompt Strategies

Alias	Description
dxgpt_standard	Standard diagnosis prompt format.
dxgpt_rare	Diagnosis prompt focused on rare diseases.
dxgpt_improved	Diagnosis prompt with thinking step & XML.
dxgpt_json	Diagnosis prompt requesting JSON output.
dxgpt_json_risk	Diagnosis prompt requesting JSON with risk handling.

Table 3: Datasets

Name	Description
RAMEDIS_200	RAMEDIS clinical cases, truncated to 200 examples.
PUMCH_ADM	PUMCH admission notes dataset.
URG_200	6500 URG Emergency room reports, truncated to 200 examples.
URG_1000	same, truncated to 1000 examples
v2	200 Synthetically generated rare disease cases (v2).

ENGINEERING CHALLENGES

Attempting to manage this combinatorial complexity using the approach adopted in the `dxgpt-testing-main` project quickly exposed significant structural and operational flaws. The ad-hoc methods, while functional for small initial tests, proved unsustainable and inefficient at scale. The key problems encountered are summarized below (Table 4):

Table 4: Key Flaws Identified in the `dxgpt-testing-main` Project

Problem	Description	Consequences
Cumbersome Workflow & Tracking	Reliance on intermediate files for data transfer and manual code changes between runs.	Hinders accurate result reproducibility.
Difficult Scalability	Adding new models, prompts, or datasets required significant code duplication and manual modification.	Limits scope of investigation; prevents thorough comparison.
Inflexible Configuration	Settings hardcoded or scattered across code, comments, and env variables, lacking a central system.	Impairs systematic exploration; risks inconsistent parameters.
Poor Code Structure	Logic tightly coupled, duplicated across files, and mixed concerns within single scripts.	Increases risk of bugs affecting results; hinders verification.
Lack of Standardization	Absence of common architectural patterns, coding conventions, or workflow structure across scripts.	Reduces clarity of methods; hinders collaboration/verification.

DATA & KNOWLEDGE SCIENCE CONSIDERATIONS

Evaluating large language models for differential diagnosis generation from biomedical cases with gold-standard annotations requires structured data preparation. Several conceptual treatments (summarized in table 5) are necessary to ensure that the evaluation measures true clinical reasoning rather than artefacts introduced by data variability.

Table 5: Standard treatments in data science

Concept	Definition (with example)	Avoids (English)
Concept Normalization	Different clinical terms that refer to the same underlying condition are mapped to a single standardized concept using controlled vocabularies like HPO or ICD-10. For example, "heart attack," and "AMI" would all be mapped to the same standarized concept "acute myocardial infarction". This prevents the model from being penalized when it uses a correct synonym.	False mismatches due to synonyms or phrasing differences.
Standardization	Data formats, coding conventions, and naming practices are unified across the dataset to prevent technical inconsistencies. For example, diagnoses from different hospitals might use slightly different code versions or structures; standardization ensures they are made uniform before evaluation. This allows fair comparison between model outputs and reference diagnoses.	Errors caused by inconsistent representations.
Named Entity Disambiguation - NED	Clinical terms that have multiple meanings are clarified based on context to ensure the correct interpretation. For instance, "MI" might mean "myocardial infarction" or "mitral insufficiency" depending on the case; disambiguation explicitly selects the correct intended meaning. This ensures that correct predictions are not wrongly evaluated as errors.	Incorrect scoring due to ambiguous terms.
Ontology Hierarchical Expansion	The evaluation allows partial matches when the model predicts a broader but clinically related concept. For example, if the gold-standard diagnosis is "hepatitis E" and the model predicts "viral hepatitis," the prediction can still be considered partially correct. This better reflects clinical reasoning,	Penalizing clinically reasonable broader predictions.

Concept	Definition (with example)	Avoids (English)
	where general categories are sometimes acceptable.	
Class Imbalance	Rare diagnoses are proportionally represented or weighted during evaluation to prevent metrics from being dominated by common diseases. Without this, a model that only predicts frequent conditions could appear highly accurate despite poor performance on rare but critical diagnoses. For example, rare genetic disorders should be properly considered alongside common infections.	Rewarding models only for predicting common conditions.

Unfortunately, these data treatments were not applied to the datasets listed in Table 3 during previous evaluations. This omission impacts the validity of the measured differential diagnosis accuracy for the LLMs. For instance, a preliminary analysis of the `PUMCH_ADM` dataset indicates that this lack of treatment could introduce an error rate of approximately 10%. Specifically, out of 79 cases analyzed:

* 3 cases lacking a gold-standard diagnosis due to non-existent mappings were incorrectly labeled as incorrect diagnoses (false negatives).
* All 5 cases of POEMs syndrome were systematically failed by all models, likely due to incorrect entity disambiguation and synonym handling.

In the `URG_200` and `URG_1000` datasets, the error rate was not numerically measured. However, it is likely similar or even higher, as these datasets contain cases where the gold-standard diagnosis uses medical jargon unfamiliar to LLMs (e.g., ITU, IRAVA, IRV, CRU), leading to systematic failures.

The dxgpt_main also presents data handling considerations. For instance, none of the analyzed cases has an unique identifier, that allows for easy comparison across models. Instead, data is fragmented across non-standardized I/O files. Considering that each differential diagnosis comprises at least 5 predicted diagnoses for a single case, and we have 450 potential experimental runs for a median of 200 cases, that makes nearly 0.5 million predicted diagnoses (450*200*5). Manually curating these predicted diagnoses across scattered files, remains impossible.

KERNEL29

Addressing these data handling aspects, alongside the experimental orchestration needs, is a focus of the Kernel29 project. It aims to provide a framework for LLM experimentation at scale that includes capabilities for capturing and structuring generated outputs. Within the medical domain, Kernel29 is designed to provide infrastructure for evaluate LLM

capabilities and utilizing their outputs in a systematic manner. To manage experimental scale and structure data handling, Kernel29 incorporates the following design principles:

* High-Quality Data: Standardizing clinical cases into a common format and normalizing diagnoses using the widespread ICD10 taxonomy.

* Robust Analysis: Defining new metrics to correctly analyze and compare LLM outputs against ground truth or expert evaluations.

* Modularity & Standardization: Ensuring all modules interacting with LLMs share a consistent structure and naming convention, based on a clear separation of concerns (API connections, model configs, prompts, parsers, database interactions).

* Data-Centric Approach: Emphasizing structured input and, crucially, the structured capture of LLM outputs into databases to enable analysis and downstream use of the generated knowledge.

* Abstraction: Hiding the specific implementation details of different LLMs behind a standard interface, simplifying interactions regardless of the provider (OpenAI, Anthropic, Azure, etc.).

* Scalability & Dynamic Loading: Utilizing aliases and runtime configuration to load necessary components (prompts, models, parsers), allowing easy experimentation with numerous combinations without hardcoding.

* Programmatic Prompting: Implementing a dedicated module for dynamic and systematic prompt construction, facilitating sophisticated prompt engineering experiments.

Chapter 2: Transforming Data for Reliable Evaluation with `bat29`

Evaluating Large Language Models in clinical diagnosis demands clean, consistent data. Raw clinical records, however, are typically messy, inconsistent, and unstructured, making direct use unreliable for rigorous experimentation. As highlighted in Table 5, variations in terminology, format, coding, and representation can obscure genuine model performance. The `src/bat29` module serves as the essential pre-processing pipeline within Kernel29, transforming these diverse raw inputs into the standardized, structured datasets required for trustworthy evaluation.

1. Consolidating and Structuring Clinical Narratives

A primary challenge is the heterogeneity of input formats. Hospital records often contain clinical details scattered across numerous free-text fields, while research datasets might simply list coded phenotypes without a narrative structure.

Input Example 1 (Hospital Record - `URG_Torre_Dic_2022_IA_GEN.csv` excerpt):

Motivo Consulta	Enfermedad Actual	Exploracion	Juicio Diagnóstico
Disnea.-	Paciente que es remitido en ambulancia… episodio de disnea…	Vigil, reactivo, aceptable estado general… Taquipneico en reposo…	Broncospasmo. Probable tapón mucoso.
Otalgia.-	Según refiere el paciente presenta desde hace 15 días… otalgia…	Lúcido, reactivo, buen estado general… Otoscopia derecha: CAE hiperémico…	Otitis externa derecha. Otitis media…

Input Example 2 (Research Data - Conceptual JSONL for Rare Diseases):

```
{ "Phenotype": ["HP:0001276", "HP:0001258", "HP:0000407"], "RareDisease": ["OMIM:176270"] } // HPO IDs for Hypotonia, Spasticity, Sensorineural hearing impairment; OMIM ID for Prader-Willi
```

Feeding such disparate data directly to an LLM or evaluation metric is problematic. `bat29` addresses this by: a) Reading various source formats (CSV, JSONL, etc.). b) Extracting relevant clinical information (symptoms from free text or coded lists, history, exam findings, demographics, vitals, test results). c) Synthesizing or restructuring this information into a standardized narrative format using a predefined template.

For hospital records like the URG data, it extracts demographics (`Sexo`, `EDAD`) and clinical details (`Motivo Consulta`, `Enfermedad Actual`, `Exploracion`, `Antecedentes`) and concatenates them into structured `Anamnesis`, `Exploracion`, etc., sections. Crucially, it also parses and formats available vital signs and complementary tests (`Exploracion Compl.`) into a standardized `Pruebas clinicas` section.

For coded phenotype lists (like the source for `test_PUMCH_ADM.csv`), it looks up the names corresponding to the HPO codes and integrates them into the template, typically listing the symptoms under `Anamnesis:`. To prevent potential biases from inconsistent original consultation reasons in source datasets, the `Motivo de consulta` for these synthesized cases is **intentionally standardized** to the neutral phrase: `Paciente acude a consulta para ser diagnosticado`. Placeholders are used for demographics if unavailable.

Simulated Output (`case` column derived from Input Example 1):

Row 1 (Disnea case):

```
Motivo de consulta:
Paciente acude a consulta para ser diagnosticado

Anamnesis:

Paciente Hombre de 54 años. Paciente que es remitido en ambulancia...
episodio de disnea...

Antecedentes:

No hay antecedentes

Exploracion:

Vigil, reactivo, aceptable estado general... Taquipneico en reposo...

Pruebas clinicas:

-Rapidas:

Tensión arterial: 144.0 / 75.0

Frecuencia cardiaca: 73.0

Temperatura: 36.6

Saturación de oxígeno: 95.0

-Complementarias:
Rx torax Portatil: ICT <0.5; no consolidaciones ni derrame pleural. No
neumotorax.-
```

Output Example (`case` column in `test_PUMCH_ADM.csv`

```
Motivo de consulta:
Paciente acude a consulta para ser diagnosticado

Anamnesis:

Paciente de sexo desconocido de desconocidos años. El paciente presenta los
siguientes síntomas:
 -Compulsive behaviors
 -Delayed speech and language development
 -Neonatal hypotonia
(... plus other phenotype names corresponding to HPO codes ...)

Antecedentes:

No hay antecedentes

Exploracion:

No se realiza

Pruebas clinicas:
nan
```

This dual transformation approach results in a consistent `case` column across all processed datasets (like those in `/data/tests/treatment/`), regardless of the original source format (scattered text fields or coded lists), providing a uniform input format suitable for LLMs.

2. Normalizing and Standardizing Diagnoses

Raw data presents diagnoses inconsistently – as free text, non-standard codes, or ontology identifiers.

Input Example (Hospital Record - `URG_Torre_Dic_2022_IA_GEN.csv` excerpt):

DIAG CIE	**Juicio Diagnóstico**
I50.9	ICC DESCOMPENSADA EPOC
J18.9	NEUMONIA BILATERAL
N23	CRU derecho.

Input Example (Research Data - Conceptual JSONL for Rare Diseases):

```
{ ... "RareDisease": ["OMIM:277900"] } // OMIM ID for Wilson Disease
{ ... "RareDisease": ["ORPHA:284979", "ORPHA:558"] } // ORPHA IDs for
Neonatal Marfan / Marfan
```

`bat29` leverages curated knowledge bases (built from ICD-10, OMIM, HPO, etc.) to standardize these inputs:

a) It maps provided codes (like `DIAG CIE`) or text diagnoses to canonical codes (primarily ICD-10 for URG, or retaining OMIM/ORPHA).

b) It retrieves the official name for the canonical code.

c) For research data, it resolves multiple IDs to a primary concept and fetches validated synonyms.

d) It combines the standard name with the original text or synonyms for completeness.

Output Example (Diagnosis columns in `test_death.csv`):

id	golden_diagnosis	diagnostic_code/s	icd10_diagnosis_name
6230	Heart failure, unspecified also known as ICC DESCOMPENSADA / EPOC	I50.9	Heart failure, unspecified
6231	Pneumonia, unspecified organism also known as NEUMONIA BILATERAL	J18.9	Pneumonia, unspecified organism

Output Example (Diagnosis columns in `test_PUMCH_ADM.csv`):

id	case	golden_diagnosis	diagnostic_code/s
0	…	Prader-Willi syndrome	[OMIM:176270]
5	…	Wilson disease also known as Hepatolenticular…	[OMIM:277900]
15	…	Neonatal Marfan syndrome also known as Marfan…	[ORPHA:284979, ORPHA:558]

This yields consistently formatted golden_diagnosis and diagnostic_code/s columns, crucial for unambiguous evaluation against ground truth.

3. Extracting Ontological Hierarchy

Raw data typically lacks the context of where a specific diagnosis fits within a broader medical classification. This hierarchical information is vital for analyzing dataset composition (e.g., prevalence of rare vs. common disease categories) and enabling evaluation metrics that understand semantic relationships.

`bat29` addresses this by using its knowledge bases to retrieve the full hierarchical path for standardized codes like ICD-10.

Output Example (Hierarchy columns in `test_death.csv`):

...	Medical speciality	Medical subspeciality	Disease group	disease group name	disease
...	IX	I30-I5A	I50	Heart failure	I50.9
...	X	J09-J18	J18	Pneumonia, unspecified organism	J18.9
...	X	J95-J99	J96	Respiratory failure, n.e.c.	J96.9

Disease name	Disease variant	Disease variant name	Disease subvariant	Disease subvariant name
Heart failure, unspecified	...	...	...	...
Pneumonia, unspecified organism	...	...	...	...
Respiratory failure, unspecified	J96.90	Resp fail, unspec w hypoxia/hypercapnia	...	...

This explicit hierarchy embedded in the output data allows for powerful analysis of case distribution across different medical specialties or disease groups (see below).

4. Generating and Analyzing Standardized Test Sets

The standardized and enriched data produced by the `bat29` pipeline serves as the foundation for creating targeted test sets for evaluating LLMs. Each test set represents a collection of clinical cases, formatted consistently and sharing a defined characteristic,

allowing for focused assessment of model capabilities across different clinical scenarios or data sources.

4.1 Test Set Creation Methodology

Specific test sets are generated by filtering the processed data based on criteria relevant to clinical outcomes, patient demographics, or data origin. This is primarily handled by scripts within the `src/bat29/` directory:

- **Outcome/Severity-Based Filtering (`treatment_urg.py`):** This script processes the standardized URG hospital records. It examines fields such as `motivo_alta_ingreso` (discharge reason, including "Fallecimiento" for death), `est_planta` (days in ward), and `est_uci` (days in ICU) to categorize cases. For instance, cases ending in death are assigned to the `death` set, while cases involving ICU stays or extended hospitalization (`>= 18` days) form the `critical` set. Cases with moderate hospitalization (`5-17` days, no ICU) constitute the `severe` set. Pediatric cases (`edad <= 15`) are also identified. Subsets limited to the first 1000 entries are created for baseline comparisons (e.g., `first_1000`, `first_1000_severe`).
- **Source-Based Processing (`treatment_ramebench_paper.py`):** This script handles data derived from research datasets focused on rare diseases (e.g., HMS, LIRICAL, MME, PUMCH_ADM). It standardizes the narrative structure (often using phenotype names derived from HPO codes) and diagnosis (using OMIM/ORPHA IDs and synonyms), creating separate test sets for each source dataset (e.g., `test_PUMCH_ADM.csv`) and a combined set (`test_ramebench.csv`).

The result of these processes is a suite of `.csv` files located in `/data/tests/treatment/`, each containing standardized `case`, `golden_diagnosis`, `diagnostic_code/s`, and ICD-10 hierarchy columns, ready for use in evaluation protocols.

4.2 Overview of Generated Test Sets

The methodology yields diverse test sets, summarized in the table below:

Test Set Name	Creation Criteria	Number of Cases
`test_all.csv`	All processed URG cases	6255
`test_1000.csv`	First 1000 processed URG cases	1000
`test_death.csv`	URG cases resulting in death	7
`test_critical.csv`	URG cases with death, ICU stay, or ward stay >= 18 days	43
`test_severe.csv`	URG cases with ward stay 5-17 days, no ICU	82
`test_pediatric.csv`	URG cases with age <= 15	1653
`test_1000_pediatric.csv`	Pediatric cases from `test_1000.csv`	335
`test_RAMEDIS.csv`	Cases from RAMEDIS rare disease dataset	624
`test_HMS.csv`	Cases from HMS rare disease dataset	87

Test Set Name	Creation Criteria	Number of Cases
test_LIRICAL.csv	Cases from LIRICAL rare disease dataset	369
test_MME.csv	Cases from MME rare disease dataset	40
test_PUMCH_ADM.csv	Cases from PUMCH Admission rare disease dataset	75
test_RAMEDIS_SPLIT.csv	Cases from RAMEDIS (Split) rare disease dataset	200
test_ramebench.csv	Combination of all processed rare disease datasets (RAMEDIS, HMS, LIRICAL, etc.)	1395

4.3 Dataset Composition Analysis

The embedded ICD-10 hierarchy enables detailed analysis of the diagnostic composition for each URG-derived test set using interactive sunburst plots (available in `/data/tests/treatment/icd10_stats/plots`). These plots visualize the distribution of cases across ICD-10 chapters, blocks, categories, and potentially finer levels.

1) `test_all` (N=6,255):

This dataset represents the most comprehensive collection of processed URG Emergency Department encounters (N=6,255) and serves as the primary baseline for assessing the general diagnostic capabilities of Large Language Models. Its composition reflects the broad clinical heterogeneity typical of unsorted emergency presentations before applying outcome or demographic filters. The Analysis of the diagnostic distribution using the embedded ICD-10 hierarchy reveals a distinct profile heavily weighted towards common, lower-to-moderate acuity conditions. The table below summarizes the prevalence of the three most frequent Primary Medical Specialties (ICD-10 Chapters) and other specialties particularly relevant to the dataset's characterization, including dominant or illustrative sub-categories:

Concept Level	ICD-10	Description	Prevalence (%)
Medical Specialty: Respiratory	**X**	**(Chapter Total)**	**~36.1%**
↳ Sub Specialty: Resp. URI	J00-J06	Acute Upper Respiratory Infections	~22.6%
↳ Disease Group: Pneumonia	J18	Pneumonia, unspecified organism	~1.3%
Medical Specialty: Injury/Poison	**XIX**	**(Chapter Total)**	**~12.9%**
↳ Sub Specialty: Ankle/Foot Inj.	S90-S99	Injuries to Ankle/Foot	~3.1%
↳ Sub Specialty: Wrist/Hand Inj.	S60-S69	Injuries to Wrist/Hand	~2.9%
↳ Disease Group: Sprain/Disloc	S93/S63	Sprains/Dislocations (Ankle/Hand)	~1.8% / ~0.7%
Medical Specialty: Musculoskeletal	**XIII**	**(Chapter Total)**	**~9.7%**
↳ Disease Group: Dorsalgia	M54	Dorsalgia (Back pain)	~3.4%
Medical Specialty: Symptoms/Signs	**XVIII**	**(Chapter Total)**	**~8.1%**
↳ Disease Group: Abd/Pelv Pain	R10	Abdominal/Pelvic Pain	~1.8%
↳ Disease Group: Headache	R51	Headache	~0.6%

Concept Level	ICD-10	Description	Prevalence (%)
↳ Disease Group: Fever	R50	Fever of unknown origin	~0.5%
Medical Specialty: Circulatory	**IX**	**(Chapter Total)**	**~1.3%**
↳ Disease Group: Heart Failure	I50	Heart Failure	~0.2%
↳ Sub Specialty: Ischemic Heart Dz	I20-I25	Ischemic Heart Diseases	~0.1%

(Note: Percentages are approximate. Sub-categories shown are illustrative.)

Conclusion & Implications for Evaluation: The quantitative profile, detailed above, highlights that `test_all` primarily tests an LLM's diagnostic breadth across common conditions encountered in primary or urgent care settings. The distribution confirms a focus on common respiratory infections, minor injuries, back pain, and frequent non-specific symptoms like pain and fever. This pattern contrasts sharply with the significant underrepresentation of high-acuity conditions, particularly within Circulatory Diseases, where specific critical categories like Heart Failure and Ischemic Heart Disease have very low prevalence. The notable proportion of cases falling under Symptoms, Signs, and Abnormal Findings (Chapter XVIII) underscores the presence of diagnostic ambiguity. Evaluating performance on these less specific presentations is crucial for assessing an LLM's ability to handle the uncertainty inherent in real-world clinical encounters. Therefore, performance on `test_all` reflects a model's ability to manage a high volume of varied, moderate-acuity cases involving common infections, minor injuries, and non-specific symptoms, rather than its proficiency in diagnosing critical or complex conditions prevalent in specialized or intensive care.

2) `test_1000` (N=1000):

This dataset consists of the first 1,000 processed URG encounters, serving as a computationally less demanding, yet substantial, random subsample of `test_all`. Its primary utility lies in providing a representative benchmark for initial or comparative evaluations where using the full `test_all` dataset may be prohibitive. As expected from a large subsample, its diagnostic composition closely mirrors that of the parent dataset.

Concept Level	ICD-10	Description	Prevalence (%)
Medical Specialty: Respiratory	**X**	**Respiratory Diseases**	**~39.1%**
Medical Specialty: Injury/Poison	**XIX**	**Injury, Poisoning, External**	**~10.1%**
Medical Specialty: Musculoskeletal	**XIII**	**Musculoskeletal Diseases**	**~10.0%**
Medical Specialty: Symptoms/Signs	**XVIII**	**Symptoms, Signs, Abnormal Findings**	**~7.6%**

Concept Level	ICD-10	Description	Prevalence (%)
Medical Specialty: Circulatory	**IX**	**Circulatory Diseases**	**~1.2%**

(Note: Percentages are approximate.)

The quantitative profile confirms the similarity to `test_all`, dominated by respiratory, injury/poisoning, musculoskeletal, and symptom-based presentations, with a similarly low prevalence of circulatory diseases. Therefore, its implications for LLM evaluation are largely identical to `test_all`: it primarily assesses diagnostic breadth across common, moderate-acuity conditions and the ability to handle diagnostic ambiguity, rather than performance on critical care or highly specialized cases.

3) `test_severe` (N=82):

This dataset focuses on cases requiring moderate hospitalization (5-17 days ward stay, no ICU), aiming to represent conditions more serious than typical ED discharges but less critical than those requiring intensive care. It is filtered from the broader URG dataset (N=82).

Concept Level	ICD-10	Description	Prevalence (%)
Medical Specialty: Respiratory	**X**	**(Chapter Total)**	**~41.5%**
↳ Disease Group: Pneumonia	J18	Pneumonia, unspecified organism	~15.9%
↳ Disease Group: Resp.Fail/Oth	J96/J98	Resp. Failure/Other Disorders	~7.3% / ~11.0%
Medical Specialty: Genitourinary	**XIV**	**(Chapter Total)**	**~12.2%**
↳ Disease Group: Renal Colic	N23	Renal Colic, unspecified	~2.4%
Medical Specialty: Digestive	**XI**	**(Chapter Total)**	**~11.0%**
↳ Disease Group: Diverticular Dz	K57	Diverticular Disease	~3.7%
↳ Disease Group: Pancreatitis	K85	Acute Pancreatitis	~1.2%
Medical Specialty: Special Purpose	**XXII**	**(Chapter Total)**	**~8.5%**
↳ Disease Group: COVID-19	U07	Emergency Use (e.g., COVID-19)	~8.5%
Medical Specialty: Injury/Poison	**XIX**	**(Chapter Total)**	**~8.5%**
Medical Specialty: Circulatory	**IX**	**(Chapter Total)**	**~4.9%**
↳ Disease Group: CVA	I63	Cerebral Infarction	~3.7%
Medical Specialty: Symptoms/Signs	**XVIII**	**(Chapter Total)**	**~4.9%**

(Note: Percentages are approximate. N=82 is relatively small. Sub-categories shown are illustrative.)

The quantitative profile of `test_severe`, detailed in the table, reflects the filtering criteria for moderate illness severity requiring inpatient care. The distribution shifts significantly compared to `test_all`, with increased prevalence of conditions often necessitating admission. While **Respiratory Diseases (X)** remain dominant, the driving categories are now more severe conditions like Pneumonia and Respiratory Failure/Disorders. Similarly, **Genitourinary (XIV)** and **Digestive Diseases (XI)** feature more prominently, represented by conditions such as renal colic, diverticular disease, and pancreatitis. The significant presence of **Codes for Special Purposes (XXII)** likely reflects specific events (e.g., COVID-19). Compared to `test_all`, the proportion of ambulatory complaints and non-specific **Symptoms/Signs (XVIII)** is reduced. Although **Circulatory Diseases (IX)** show only a modest increase overall, the presence of conditions like Cerebral Infarction points to higher acuity within this subset. This dataset thus challenges LLMs on diagnoses associated with moderate illness severity requiring hospital management, representing a distinct step up in complexity from the broad baseline but stopping short of critical care scenarios.

4) `test_critical` (N=43):

This dataset specifically targets high-acuity cases, including those resulting in death, requiring an ICU stay, or involving prolonged hospitalization (>= 18 days). It represents a small (N=43) but clinically significant subset focused on life-threatening conditions.

Concept Level	ICD-10	Description	Prevalence (%)
Medical Specialty: Respiratory	**X**	**(Chapter Total)**	**~37.2%**
↳ Disease Group: Pneumonia	J18	Pneumonia, unspecified organism	~16.3%
↳ Disease Group: Resp.Fail/Oth	J96/J98	Resp. Failure/Other Disorders	~7.0% / ~7.0%
Medical Specialty: Circulatory	**IX**	**(Chapter Total)**	**~18.6%**
↳ Disease Group: Heart Failure	I50	Heart Failure	~9.3%
↳ Disease Group: CVA	I63	Cerebral Infarction	~2.3%
Medical Specialty: Digestive	**XI**	**(Chapter Total)**	**~14.0%**
↳ Disease Group: Cholecystitis	K81	Cholecystitis	~4.7%
↳ Disease Group: Ileus	K56	Paralytic Ileus/Obstruction	~4.7%
Medical Specialty: Genitourinary	**XIV**	**(Chapter Total)**	**~7.0%**
Medical Specialty: Symptoms/Signs	**XVIII**	**(Chapter Total)**	**~7.0%**
Medical Specialty: Injury/Poison	**XIX**	**(Chapter Total)**	**~7.0%**

(Note: Percentages are approximate and based on a small N=43. Sub-categories shown are illustrative.)

The diagnostic profile of `test_critical`, detailed above, clearly reflects its focus on severe illness. The high prevalence of **Respiratory Diseases (X)** is characterized by conditions like Pneumonia and Respiratory Failure. Crucially, **Circulatory Diseases (IX)** constitute a much larger proportion compared to less severe subsets, driven primarily by critical conditions such as Heart Failure, along with other serious vascular events. **Digestive Diseases (XI)** also feature prominently with severe conditions like cholecystitis and ileus. The relative contributions from less specific **Symptoms/Signs (XVIII)** and **Injury (XIX)** are lower than in the baseline, suggesting a concentration on more clearly defined, life-threatening systemic diseases. Evaluating LLMs on this dataset specifically probes their ability to recognize and diagnose high-acuity conditions where timely and accurate assessment is paramount, offering a distinct challenge focused on critical care knowledge (particularly related to respiratory and cardiac failure) rather than diagnostic breadth.

5) `test_death` (N=7):

This dataset represents an extremely small (N=7) and highly specific subset containing only cases from the URG dataset that resulted in patient death. Its purpose is to isolate the diagnostic patterns associated with fatal outcomes within this cohort.

Concept Level	ICD-10	Description	Count (N=7)
Medical Specialty: Circulatory	**IX**	**Circulatory Diseases**	**2**
Medical Specialty: Respiratory	**X**	**Respiratory Diseases**	**2**
Medical Specialty: Digestive	**XI**	**Digestive Diseases**	**2**
Medical Specialty: Symptoms/Signs	**XVIII**	**Symptoms, Signs, Abnormal Findings**	**1**

(Note: Due to N=7, counts are shown instead of percentages.)

Given the extremely small sample size, the diagnostic profile is highly concentrated among conditions directly associated with mortality in this specific cohort. **Circulatory (IX)**, **Respiratory (X)**, and **Digestive Diseases (XI)** each account for two cases, with one case coded under **Symptoms/Signs (XVIII)**. While drawing broad conclusions is impossible due to the low N, this distribution aligns with common causes of in-hospital mortality (e.g., severe cardiac events, respiratory failure, complications of digestive system diseases). Evaluating an LLM on this dataset tests its ability to recognize potentially fatal conditions, although the statistical significance is minimal. It primarily serves as a qualitative indicator of model performance on the most severe end of the clinical spectrum represented in the source data.

6) `test_pediatric` (N=1653):

This large dataset (N=1653) includes all processed URG encounters for patients aged 15 years or younger. It aims to specifically evaluate LLM performance on common pediatric presentations seen in an emergency setting.

Concept Level	ICD-10	Description	Prevalence (%)
Medical Specialty: Respiratory	**X**	**(Chapter Total)**	**~55.8%**
↳ Sub Specialty: Resp. URI	J00-J06	Acute Upper Respiratory Infections	~37.1%
↳ Disease Group: Bronchiolitis	J21	Acute Bronchiolitis	~3.0%
↳ Disease Group: Pneumonia	J18	Pneumonia, unspecified organism	~1.8%
Medical Specialty: Injury/Poison	**XIX**	**(Chapter Total)**	**~12.2%**
↳ Disease Group: Sprain/Disloc	S93/S63	Sprains/Dislocations (Ankle/Hand)	~1.8% / ~1.1%
↳ Disease Group: Superficial Inj	S60/S90	Superficial Injuries (Hand/Foot)	~1.5% / ~1.0%
Medical Specialty: Ear	**VIII**	**(Chapter Total)**	**~8.7%**
↳ Disease Group: Otitis Media	H66	Otitis Media, supp./unspec.	~7.2%
Medical Specialty: Symptoms/Signs	**XVIII**	**(Chapter Total)**	**~6.0%**
↳ Disease Group: Fever	R50	Fever of unknown origin	~1.3%
↳ Disease Group: Abd/Pelv Pain	R10	Abdominal/Pelvic Pain	~1.8%
Medical Specialty: Infectious	**I**	**(Chapter Total)**	**~4.3%**
↳ Disease Group: Viral Inf	B34	Viral infection, unspecified	~2.3%

(Note: Percentages are approximate. Sub-categories shown are illustrative.)

The diagnostic profile of `test_pediatric`, detailed in the table, clearly reflects common childhood ailments presenting to emergency care. **Respiratory Diseases (X)** are overwhelmingly dominant, largely attributable to high rates of various Acute Upper Respiratory Infections and specific lower respiratory conditions like Acute Bronchiolitis. **Injury/Poisoning (XIX)** primarily involves minor extremity trauma. **Diseases of the Ear (VIII)** are notably prevalent, driven overwhelmingly by Otitis Media. **Symptoms/Signs (XVIII)**, particularly Fever and Abdominal Pain, and non-specific **Infectious Diseases (I)** are also significant contributors. Conversely, specialties common in adults, such as Musculoskeletal (XIII) and especially Circulatory (IX), are much less represented. This dataset provides a valuable benchmark for assessing LLM diagnostic accuracy specifically

within the pediatric domain, testing knowledge of common childhood conditions like URIs, bronchiolitis, otitis media, minor trauma, and undifferentiated febrile presentations.

7) `test_1000_pediatric` (N=335):

This dataset is a subset of `test_1000`, containing only the pediatric cases (age <= 15) from those first 1,000 encounters (N=335). It allows for analysis of pediatric presentations within the specific context of the initial data sample.

Concept Level	ICD-10	Description	Prevalence (%)
Medical Specialty: Respiratory	**X**	**Respiratory Diseases**	**~64.2%**
Medical Specialty: Ear	**VIII**	**Ear Diseases**	**~9.6%**
Medical Specialty: Injury/Poison	**XIX**	**Injury, Poisoning, External**	**~5.4%**
Medical Specialty: Symptoms/Signs	**XVIII**	**Symptoms, Signs, Abnormal Findings**	**~4.2%**
Medical Specialty: Infectious	**I**	**Infectious Diseases**	**~3.9%**

(Note: Percentages are approximate.)

As a pediatric subset of `test_1000`, this dataset's profile strongly resembles `test_pediatric` but reflects the specific case mix within the initial 1000 URG records. **Respiratory Diseases (X)** are even more pronounced here (~64.2%) than in the overall pediatric cohort. **Ear Diseases (VIII)** (~9.6%) remain highly prevalent. **Injury/Poisoning (XIX)** (~5.4%) is present but represents a smaller fraction compared to the full `test_pediatric` set. **Symptoms/Signs (XVIII)** (~4.2%) and **Infectious Diseases (I)** (~3.9%) contribute moderately. The implications for evaluation are similar to `test_pediatric`, focusing on common childhood conditions, particularly respiratory and ear-related ailments, but derived from a smaller, specific cross-section of the overall URG data.

Examining the specific sunburst plots for each set provides quantitative confirmation of these expected distributions and highlights the specific diagnostic challenges inherent in each subset, allowing researchers to select appropriate tests for targeted LLM evaluation.

4.4 Future Directions for Test Set Creation

The standardized framework facilitates the creation of further specialized test sets to probe LLM performance more deeply:

- **Domain-Specific:** Filtering by `icd10_chapter_code` to create sets for cardiology (Chapter IX), pulmonology (Chapter X), etc.
- **Demographically Stratified:** Creating subsets balanced by age or sex to investigate potential biases.

- **Symptom-Oriented:** Developing sets based on primary presenting symptoms, if reliably extractable.
- **Hybrid Sets:** Combining cases from different sources (e.g., URG pediatric and Ramebench pediatric) for broader scope.
- **Refined Outcome Sets:** Creating sets excluding symptom-based codes (Chapter XVIII) to focus on definitive diagnoses.

Such targeted datasets will be invaluable for nuanced and robust evaluation of clinical diagnostic LLMs.

Chapter 2 Appendix: Technical details

This appendix provides a technical description of the Python scripts located within the `src/bat29` directory. These scripts are responsible for reconstructing datasets, building knowledge bases, treating raw data, loading processed data into the database, and performing initial analysis and visualization for the Kernel29 benchmark datasets.

1. Dataset Reconstruction

These scripts aim to recover and assemble benchmark datasets, missing from the previous dxgpt-main framework from various source files and formats.

`reconstruct_RAMEDIS_split.py`

Purpose

To download the `RAMEDIS` dataset test split from the Hugging Face `chenxz/RareBench` repository, select the first 200 examples, and save them into a JSON Lines (`.jsonl`) file.

General Technical Overview

1. Loads the `RAMEDIS` test split from `chenxz/RareBench` using the `datasets` library.
2. Selects the first 200 entries.
3. Saves these 200 examples to `RAMEDIS_SPLIT.jsonl`.

`reconstruct_PUMCH_ADM_part1.py`

Purpose

To reconstruct the initial part of the `PUMCH_ADM` dataset by consolidating patient information stored in individual JSON files into a single CSV file.

General Technical Overview

1. Finds all `patient_*.json` files in the specified input directory.
2. Reads each JSON file, extracts patient information, and adds the patient number.
3. Consolidates the data into a pandas DataFrame.
4. Sets the patient number as the index and sorts.
5. Saves the DataFrame to `PUMCH_ADM_reconstructed_part1.csv`.

`reconstruct_PUMCH_ADM_part2.py`

Purpose

To combine the partially reconstructed `PUMCH_ADM` data (from Part 1) with corresponding scores and existing knowledge base mappings to create a final `.jsonl` file containing Phenotype (HPO IDs) and RareDisease (OMIM Disease IDs) for each case. It also generates an auxiliary mapping file (`disease2name_juanjo.json`).

General Technical Overview

1. Loads prerequisite mapping files (disease, HPO) and the Part 1 CSV and external scores CSV.
2. Aligns the data from the reconstructed data and scores files.
3. Iterates through cases:
    - Extracts and validates disease names from scores, maps to OMIM IDs.
    - Updates an auxiliary disease name mapping.
    - Extracts phenotype names from reconstructed data, maps to HPO IDs.
    - Creates a result dictionary with "Phenotype" (HPO IDs) and "RareDisease" (OMIM IDs).
4. Saves the list of result dictionaries to `PUMCH_ADM_reconstructed.jsonl`.
5. Saves the updated auxiliary disease name mapping.

2. Knowledge Base Construction

These scripts process raw knowledge source files to generate structured, computationally accessible mapping files (Python dictionaries or JSON) used for standardization and analysis.

`construct_kb_ICD10_part1.py`

Purpose

To parse a raw text file (`icd10_chapters_blocks`) containing ICD-10 chapter titles (Roman numerals) and block code ranges (e.g., `A00-A09`) and create an initial mapping from base ICD-10 codes (e.g., 'A00', 'B25') to their corresponding chapter and block descriptions.

General Technical Overview

1. Reads the raw `icd10_chapters_blocks` text file.
2. Parses lines to identify ICD-10 chapter titles and block code ranges.
3. Expands block ranges (e.g., `A00-A09`) into individual base codes (e.g., `A00`, `A01`, … `A09`).
4. Creates a dictionary mapping each base code to its chapter and block description.
5. Applies manual corrections for specific codes and standardizes descriptions.
6. Saves the resulting mapping dictionary to `code2mappings.py`.

`construct_kb_ICD10_part2.py`

Purpose

To parse the detailed ICD-10 CM order file (`icd10cm-order-April-2025.txt`) to build comprehensive mappings from specific ICD-10 codes (including sub-codes like `A01.0`) to their official descriptive names and their full hierarchical path (chapter, block, category, etc.).

General Technical Overview

1. Reads the detailed ICD-10 CM order file (`icd10cm-order-April-2025.txt`).
2. Imports the base code mapping (`code2mappings`) created in Part 1.

3. Parses each line to extract the specific ICD code, hierarchy level, and official description.
4. For each code:
    - Uses the base code to find chapter/block info from `code2mappings`.
    - Determines the full hierarchical path (chapter, block, category, etc.) based on code structure.
    - Builds a mapping from the specific code to its official name (`dcode2names`).
    - Builds a mapping from the specific code to its full hierarchical path details (`dcode2parents`).
5. Saves the name mapping to `icd10_code2names.py`.
6. Saves the hierarchy mapping to `icd10_code2branch.py`.

`construct_kb_OMIM_HPO.py`

Purpose

To construct multiple mapping files related to OMIM diseases and HPO terms, primarily focusing on linking disease IDs, names, synonyms, and HPO IDs.

General Technical Overview

1. Loads existing HPO ID -> Name and OMIM Disease ID -> Primary Name mappings.
2. Creates a reverse Name -> HPO ID mapping.
3. Iterates through the primary disease names:
    - Normalizes and validates the name to generate a list of synonyms.
    - Creates mappings: Synonym -> OMIM ID (`name2disease`).
    - Creates mappings: OMIM ID -> List of Synonyms (`disease2synonyms`).
    - Creates mappings: Extended Name String -> OMIM ID (`name2disease_extended`).
4. Saves the generated mappings (`disease2synonyms`, `name2hpo`, `name2disease`, `name2disease_extended`) as JSON files.

3. Dataset Treatment

These scripts apply cleaning, normalization, and standardization rules to specific raw or reconstructed datasets, preparing them for loading into the database.

Helper Functions for Case Standardization (`utils.helper_functions.py`)

Several scripts, particularly `treatment_urg.py` and `treatment_ramebench_paper.py`, rely on helper functions defined in `src/bat29/utils/helper_functions.py` to construct a standardized clinical case text from diverse input fields. These functions ensure a consistent format for subsequent processing or LLM input.

- `do_motivo_consulta(motivo_consulta=None)`: ### Purpose Formats the “Reason for Consultation” section.

General Technical Overview

- Returns a standard header “Motivo de consulta:”, appending the provided text or a default phrase if none is given.
- `do_anamnesis(sexo=None, edad=None, enfermedad_actual=None)`: ### Purpose Formats the “Anamnesis” (Patient History) section.

General Technical Overview

- Creates a standard header “Anamnesis:de [edad] años.” and appends the `enfermedad_actual` text (often containing phenotype details).
- `do_antecedentes(antecedentes=None)`: ### Purpose Formats the “Antecedentes” (Medical History/Background) section.

General Technical Overview

- Returns a standard header “Antecedentes:”, appending the provided text or “No hay antecedentes” if none is given.
- `do_exploracion(exploracion)`: ### Purpose Formats the “Exploracion” (Physical Examination) section.

General Technical Overview

- Returns a standard header “Exploracion:” followed by the provided examination text.
- `do_pruebas(ta_max=None, ta_min=None, frec_cardiaca=None, temperatura=None, sat_oxigeno=None, glucemia=None, diuresis=None, exploracion_compl=None)`: ### Purpose Formats the “Pruebas clinicas” (Clinical Tests/Vitals) section, separating rapid tests (vitals) from complementary tests.

General Technical Overview

1. Starts with the “Pruebas clinicas:” header.
2. Safely converts vital sign inputs to numbers.
3. If vital signs exist, adds a “-Rapidas:” sub-header and lists formatted vitals.
4. If complementary tests (`exploracion_compl`) exist, adds a “-Complementarias:” sub-header and appends the text.
5. Combines the sections, omitting sub-headers if no data exists.

- `do_diagnostico(juicio_diagnostico, icd10_code=None, icd10code2name=None, force_exit=True)`: ### Purpose Formats the final diagnosis string, potentially combining clinician’s text with standardized ICD-10 information or handling lists of synonyms.

General Technical Overview

1. Handles input `juicio_diagnostico`: formats lists of synonyms or converts other types to string.
2. If no ICD-10 code is provided or the clinician’s diagnosis is empty, returns the formatted `juicio_diagnostico`.
3. If an ICD-10 code is provided:
    - Looks up the official ICD-10 name.

- Checks if the clinician's diagnosis represents multiple comorbid conditions or a synonym for the ICD-10 name.
- Returns a formatted string indicating comorbidity ("Comorbid diagnosis…"), synonymy ("[ICD10 Name] also known as [Clinician Diagnosis]"), or just the official ICD-10 name if they match or the clinician's text was empty.

- `do_case(motivo_consulta=None, anamnesis=None, antecedentes=None, exploracion=None, pruebas=None)`: ### Purpose Concatenates the formatted sections generated by the other `do_*` functions into a single clinical case text string.

General Technical Overview

- Joins the formatted strings for Motivo, Anamnesis, Antecedentes, Exploracion, and Pruebas with "" separators.

`treatment_urg.py`

Purpose

To process a specific Excel file (`URG_Torre_Dic_2022_IA_GEN.xlsx`) containing emergency room reports, clean the data, construct a standardized case text format, map diagnoses to ICD-10 codes using the generated knowledge base, extract ICD-10 hierarchy details, and save the processed data into multiple filtered CSV files based on criteria like severity or patient age.

General Technical Overview

1. Loads ICD-10 mappings and the input Excel file (`URG_Torre_Dic_2022_IA_GEN.xlsx`).
2. Iterates through each row (emergency room report):
   - Extracts relevant patient and clinical fields.
   - Uses helper functions (`do_anamnesis`, `do_pruebas`, etc.) to assemble a standardized case text (`caso`).
   - Uses `do_diagnostico` to format the golden diagnosis string, incorporating the ICD-10 code.
   - Looks up detailed ICD-10 hierarchy (chapter, block, etc.) using `get_icd10_details` and mappings.
   - Stores the processed data (ID, case text, golden diagnosis, ICD-10 details) in a temporary structure.
   - Appends the processed data to lists corresponding to predefined filters (e.g., death, critical, pediatric, all).
3. Saves each filtered list to a separate CSV file (e.g., `test_critical.csv`, `test_pediatric.csv`, `test_all.csv`).

`treatment_ramebench_paper.py`

Purpose

To process several rare disease benchmark datasets (RAMEDIS, HMS, LIRICAL, MME, PUMCH_ADM, RAMEDIS_SPLIT) originally in `.jsonl` format, extract phenotype and disease

information, convert them into a standardized text case format similar to `treatment_urg.py`, format the golden diagnosis using synonyms, and save each processed dataset as a separate `test_*.csv` file, plus a combined `test_ramebench.csv`.

General Technical Overview

1. Loads knowledge base mappings (disease synonyms, HPO names).
2. Loads multiple rare disease `.jsonl` datasets (RAMEDIS, HMS, etc.).
3. Iterates through each dataset:
    - For each case (line) in the dataset:
        - Extracts OMIM Disease IDs and HPO Phenotype IDs.
        - Looks up disease synonyms and HPO names using mappings.
        - Constructs a standardized case text (`caso`) using helper functions, placing HPO names in the "Enfermedad Actual" section.
        - Constructs a golden diagnosis string (`golden_case`) using `do_diagnostico` with the list of disease synonyms.
        - Stores the processed data (index, case text, golden diagnosis, OMIM IDs).
    - Saves the processed data for the current dataset to a `test_{dataset_name}.csv` file.
4. Saves the combined processed data from all datasets to `test_ramebench.csv`.

4. Data Loading

These scripts read the processed and standardized data files (primarily CSVs generated by the treatment scripts) and populate the corresponding Kernel29 database tables.

`load_cases.py`

Purpose

To read standardized `test_*.csv` files from specified directories and insert the core case information (case text, source identifier) into the `cases_bench` database table.

General Technical Overview

1. Identifies all `test_*.csv` files in specified input directories.
2. Establishes a database connection.
3. Iterates through each identified CSV file:
    - Reads the CSV into a pandas DataFrame.
    - Uses the filename base (e.g., `test_RAMEDIS`) as the `hospital` identifier.
    - Iterates through each row (case):
        - Extracts the case text and original row ID (used as `source_file_path`).
        - Calls `add_cases_bench` to insert the `hospital`, `original_text`, `source_type` ("test"), and `source_file_path` into the `cases_bench` table.
4. Closes the database connection.

`load_case_metadata.py`

Purpose

To read the same `test_*.csv` files processed by `load_cases.py`, extract metadata associated with each case (like diagnostic codes, ICD-10 hierarchy information, severity), link it to the correct case in the `cases_bench` table, and insert the metadata into the `cases_bench_metadata` table.

General Technical Overview

1. Identifies all `test_*.csv` files in specified input directories.
2. Defines a mapping from test filenames (e.g., `test_critical`) to severity level IDs.
3. Establishes a database connection.
4. Iterates through each identified CSV file:
    - Reads the CSV into a DataFrame.
    - Extracts the base filename (`test_name`).
    - Iterates through each row (case):
        - Extracts the original row identifier (`id`).
        - Queries the `cases_bench` table using `test_name` and `id` to find the corresponding `cases_bench_id`.
        - Determines the severity level ID using the filename mapping.
        - Extracts other metadata (diagnostic codes, ICD-10 details) from the row.
        - Calls `add_case_metadata` to insert the metadata into the `cases_bench_metadata` table, linked via `cases_bench_id`.
5. Closes the database connection.

`load_case_golden_diagnosis.py`

Purpose

To read the `test_*.csv` files, extract the golden diagnosis information for each case, link it to the correct entry in the `cases_bench` table, and insert it into the `cases_bench_diagnosis` table.

General Technical Overview

1. Identifies all `test_*.csv` files in specified input directories.
2. Establishes a database connection.
3. Iterates through the identified CSV files:
    - (Dependency Note: Attempts to load a separate `mapping_output.jsonl` file, expected to map original row IDs to `cases_bench_id`. If this file is missing, the script skips the CSV).
    - Reads the CSV into a DataFrame.
    - Loads the ID mapping from `mapping_output.jsonl`.
    - Iterates through each row (case):
        - Extracts the golden diagnosis text and diagnostic code(s).

- Uses the loaded mapping to find the `cases_bench_id` corresponding to the original row ID.
- Calls `add_case_diagnosis` to insert the golden diagnosis information into the `cases_bench_diagnosis` table, linked via `cases_bench_id`.

4. Closes the database connection.

Chapter 3: Advanced Metrics for Evaluating Differential Diagnosis Quality

Evaluating the quality of a differential diagnosis list generated by a Large Language Model (LLM) requires metrics that capture the nuances of clinical reasoning. Well established metrics like Top-1 or Top-5 accuracy, provide a fundamental assessment of correctness by verifying the presence of the correct diagnosis within the top N suggestions, they may not fully encompass the complexities inherent in constructing a clinically useful differential diagnosis. A well-formed differential list ideally not only includes the correct diagnosis but also ranks diagnoses appropriately by likelihood and ensures that alternative suggestions represent clinically relevant or plausible considerations.

This chapter introduces two advanced, weighted scoring metrics developed within Kernel29: the Semantic Relationship Score and the Severity Matching Score. These metrics aim to complement traditional accuracy measures by providing a more holistic and clinically meaningful evaluation. They achieve this by acknowledging that diagnoses ranked higher (deemed more likely) should contribute more significantly to the assessment and by factoring in the "distance" (based on semantic relationships or severity levels) between a suggested diagnosis and the true (golden) diagnosis, thereby valuing closer matches more highly than unrelated or significantly mismatched suggestions.

1. Complementing Traditional Accuracy Metrics

Traditional metrics such as Top-1 and Top-5 accuracy offer valuable baseline information by treating the differential diagnosis list as a set membership problem, determining if the single most likely diagnosis (Top-1) or any of the top N suggestions (Top-N) match the correct diagnosis. While informative, these metrics primarily focus on the presence or absence of the correct diagnosis within a specified cutoff. They do not inherently capture certain aspects that are often crucial in clinical practice. For instance, Top-N accuracy treats a correct diagnosis identified at rank 1 the same as one found at rank N, potentially overlooking the model's ability to pinpoint the most probable cause. Furthermore, these metrics may not differentiate between a list where alternative diagnoses are plausible clinical considerations (e.g., sharing pathophysiology or belonging to the same disease group) versus a list containing unrelated conditions. They also typically do not assess the alignment between the severity level implied by the suggested diagnoses and the patient's actual condition severity. Consequently, they evaluate individual list positions for correctness rather than assessing the overall clinical utility and coherence of the differential diagnosis as a whole.

To provide a more comprehensive evaluation addressing these facets, Kernel29 employs supplementary weighted scoring methods, detailed below.

2. A Holistic Weighted Scoring Approach

To incorporate rank importance and clinical relevance, a weighted scoring methodology is employed. The core idea is to assign a score to each diagnosis in the generated list based on its rank and its "distance" from the golden diagnosis, then compute a weighted average reflecting the overall quality of the list. The general formula structure is:

$$Score = \frac{\sum_{i=1}^{N}(w_i \times (D_{\max} - D_i)^2)}{\sum_{i=1}^{N} w_i}$$

Where:

- `N` represents the number of diagnoses in the LLM's output list being evaluated (typically 5).
- `i` is the rank position (index) of a diagnosis within the list, ranging from 1 (most likely) to N.
- `w_i` is the weight assigned to rank `i`, reflecting the principle that higher-ranked (more likely) diagnoses carry more importance. Higher ranks receive higher weights (e.g., using (w_i = (6-i)/5) for N=5, as shown in Table 1).
- `D_i` is the calculated "distance" measuring the mismatch (either semantic or severity-based) between the suggested diagnosis at rank `i` and the golden diagnosis. A smaller distance indicates a better match.
- `D_max` is the maximum possible distance value for the specific metric (semantic or severity). The term `(D_max - D_i)` translates the distance `D_i` into a "match score," rewarding smaller distances (better matches) with higher values. Squaring this term amplifies the reward for very close matches (low `D_i`).
- The denominator `Σ w_i` represents the sum of all weights applied for the list. This normalization ensures that the final score is an average weighted quality score, allowing for fair comparison even if list lengths (N) or the sum of weights vary (e.g., if N < 5).

Kernel29 utilizes Linear Weights for `w_i`, prioritizing higher ranks as detailed in Table 1:

Table 1:

Rank (i)	**Weight** $w_i = (6 - i)/5$
1	1.0
2	0.8
3	0.6
4	0.4
5	0.2
Sum	**3.0**

This linear weighting scheme ensures that a correct or highly relevant diagnosis at Rank 1 contributes substantially more to the final score than the same diagnosis placed at Rank 5, reflecting clinical prioritization.

3. LLM-as-Judge for Classification

To systematically obtain the semantic relationship categories (Table 2) and severity levels (Table 3) needed for the scoring calculations, an automated approach using a Large Language Model as a classifier, often termed "LLM-as-judge", was employed. Specifically, Llama 3.1 70B was utilized.

- Severity Classification: Llama 3.1 70B was used to classify the severity level for:
  - Every diagnosis predicted by the target LLM.
  - The golden diagnosis for a case, but only if the severity level was not already provided in the source dataset. In both scenarios, the LLM was prompted to classify the severity into one of the categories: `mild`, `moderate`, `severe`, `critical`, or `rare` based on the diagnosis name (a multi-class classification task).
- Semantic Relationship Classification: For each pair consisting of the golden diagnosis and a predicted diagnosis from the target LLM's list, Llama 3.1 70B was prompted to classify the semantic relationship between them. It selected one category from Table 2: `Exact Synonym`, `Broad Synonym`, `Exact Disease Group`, `Broad Disease Group`, or `Not Related`.

This automated classification provided the necessary categorical inputs (`D_i` for semantics, and the severity values used to calculate `D_i` for severity) required by the weighted scoring formulas (Equation 1). The reliability of these classifications depends on the capability of the judge LLM (Llama 3.1 70B) and the specific prompts used. As noted in the Appendix, this implementation serves as an initial example, and further refinement might be necessary for robust, production-level evaluation.

4. Combined Score Calculation Examples

To illustrate how these weighted scores capture different facets of differential diagnosis quality, this section examines three illustrative cases from the `c3opus` model evaluated on the `PUMCH_ADM` dataset (`ramedis` bench) using the `dxgpt_main` prompt. Both the Semantic Relationship Score and the Severity Matching Score will be calculated and discussed for each case.

4.1 Semantic Relationship Score Definition

The Semantic Relationship Score quantifies the clinical relatedness between the suggested diagnoses and the golden diagnosis. The distance `D_i` for this score is directly assigned based on predefined clinical relationships mapped to discrete distance values, as shown in Table 2. (The method for determining these relationships is detailed in Section 3.3).

Table 2:

Semantic Relationship	Distance (D_1)	Interpretation
Exact Synonym	1	Perfect match
Broad Synonym	2	Closely related term
Exact Disease Group	3	Same specific disease category
Broad Disease Group	4	Related broader disease category
Not Related	5	Unrelated condition

The maximum distance (`D_max`) for this score is 5, and the linear weights from Table 1 are applied.

4.2 Severity Matching Score Definition

The Severity Matching Score assesses the alignment between the severity level of the suggested diagnoses and the severity level of the golden diagnosis. Unlike the direct assignment used for semantic distance, the severity distance involves calculation based on numerical values.

Severity Values:
Both the severity of the golden diagnosis and the severity of each predicted diagnosis were determined using an automated classification approach described in Section 3.3. These clinical severity levels are mapped to ordinal numerical values as shown in Table 3.

Table 3:

Severity Level	Value
mild	1
moderate	2
severe	3
critical	4
rare	5

Severity Distance (`D_i`):
After converting the severity labels (for both the golden and predicted diagnosis at rank `i`) to their numerical values using Table 3, the distance `D_i` is calculated based on their absolute difference:

$$D_i = 1 + |\text{golden\_severity\_value} - \text{predicted\_severity\_value}|$$

This formula ensures the distance reflects the magnitude of the severity mismatch:

- An exact match (e.g., rare vs rare, value 5 vs 5) results in the minimum distance: ($D_i = 1 + |5-5| = 1$).
- The maximum mismatch (e.g., rare vs mild, value 5 vs 1) results in the maximum distance: ($D_i = 1 + |5-1| = 5$).

Maximum Distance (D_{max}):

Derived from the calculation method, the maximum possible severity distance is 5. This value is used for D_{max} in the general scoring formula (Equation 1) when calculating the Severity Score.

Weights (w_i):

The same linear weights defined in Table 1 ([1.0, 0.8, 0.6, 0.4, 0.2] for N=5) are applied.

4.3 Calculation Examples

Now, let's apply these definitions to our three cases, selected to represent best, medium, and worst overall performance based on combined scores.

4.3.1 Case 1: Best Performance (Patient ID 31)

Golden Diagnosis: "Myasthenia gravis" (Severity: rare = 5)

LLM Output (N=3):

Rank	Predicted Diagnosis	Severity	Semantic Relationship
1	Myasthenia Gravis	rare	Exact Synonym
2	Lambert-Eaton Myasthenic Syndrome (LEMS)	rare	Exact Disease Group
3	Botulism	severe	Broad Disease Group

Semantic Score Calculation (Adjusted for N=3, Sum Weights = 2.4):

Rank	Weight (w_i)	Semantic Relationship	Sem. Distance (D_i)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	Exact Synonym	1	$1.0 * (5-1)^2 = 16.0$	16.0
2	0.8	Exact Disease Group	3	$0.8 * (5-3)^2 = 3.2$	3.2
3	0.6	Broad Disease Group	4	$0.6 * (5-4)^2 = 0.6$	0.6
	Sum = 2.4			Numerator:	19.8

Semantic Score = 19.8 / 2.4 = 8.25

Severity Score Calculation (Adjusted for N=3, Sum Weights = 2.4): (Golden Severity = 5)

Rank	Weight (w_i)	Predicted Severity	Sev. Distance (D_i = 1 + |5 - pred|)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	rare (5)	1	$1.0 * (5-1)^2 = 16.0$	16.0
2	0.8	rare (5)	1	$0.8 * (5-1)^2 = 12.8$	12.8
3	0.6	severe (3)	3	$0.6 * (5-3)^2 = 2.4$	2.4
	Sum = 2.4			Numerator:	31.2

Severity Score = 31.2 / 2.4 = 13.0

4.3.2 Case 2: Medium Performance (Patient ID 54)

Golden Diagnosis: “Myofibrillar myopathy ZASP-related” (Severity: rare = 5)

LLM Output:

Rank	Predicted Diagnosis	Severity	Semantic Relationship
1	Myofibrillar Myopathy (MFM)	rare	Broad Synonym
2	Desminopathy	rare	Broad Synonym
3	Limb-Girdle Muscular Dystrophy (LGMD)	rare	Exact Disease Group
4	Inclusion Body Myositis (IBM)	severe	Broad Disease Group
5	Pompe Disease	rare	Broad Disease Group

Semantic Score Calculation:

Rank	Weight (w_i)	Semantic Relationship	Sem. Distance (D_i)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	Broad Synonym	2	$1.0 * (5-2)^2 = 9.0$	9.0
2	0.8	Broad Synonym	2	$0.8 * (5-2)^2 = 7.2$	7.2
3	0.6	Exact Disease Group	3	$0.6 * (5-3)^2 = 2.4$	2.4
4	0.4	Broad Disease Group	4	$0.4 * (5-4)^2 = 0.4$	0.4
5	0.2	Broad Disease Group	4	$0.2 * (5-4)^2 = 0.2$	0.2
	Sum = 3.0			Numerator:	19.2

Semantic Score = 19.2 / 3.0 = 6.4

Severity Score Calculation: (Golden Severity = 5)

Rank	Weight (w_i)	Predicted Severity	Sev. Distance ($D_i = 1 + |5 - pred|$)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	rare (5)	1	$1.0 * (5-1)^2 = 16.0$	16.0
2	0.8	rare (5)	1	$0.8 * (5-1)^2 = 12.8$	12.8

Rank	Weight (w_i)	Predicted Severity	Sev. Distance (D_i = 1 + |5 - pred|)	Score Calc: w_i*(5-D_i)^2	Contribution
3	0.6	rare (5)	1	0.6 * (5-1)^2 = 9.6	9.6
4	0.4	severe (3)	3	0.4 * (5-3)^2 = 1.6	1.6
5	0.2	rare (5)	1	0.2 * (5-1)^2 = 3.2	3.2
	Sum = 3.0			Numerator:	43.2

Severity Score = 43.2 / 3.0 = 14.4

4.3.3 Case 3: Worst Performance (Patient ID 20)

- Golden Diagnosis: “Brugada syndrome” (Severity: rare = 5)
- LLM Output:

Rank	Predicted Diagnosis	Severity	Semantic Relationship
1	Benign Prostatic Hyperplasia (BPH)	mild	Not Related
2	Myasthenia Gravis	rare	Not Related
3	Rhabdomyolysis	severe	Not Related
4	Syncope (Fainting)	mild	Broad Disease Group
5	Acute Coronary Syndrome (ACS)	critical	Broad Disease Group

Semantic Score Calculation:

Rank	Weight (w_i)	Semantic Relationship	Sem. Distance (D_i)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	Not Related	5	$1.0 * (5-5)^2 = 0.0$	0.0
2	0.8	Not Related	5	$0.8 * (5-5)^2 = 0.0$	0.0
3	0.6	Not Related	5	$0.6 * (5-5)^2 = 0.0$	0.0
4	0.4	Broad Disease Group	4	$0.4 * (5-4)^2 = 0.4$	0.4
5	0.2	Broad Disease Group	4	$0.2 * (5-4)^2 = 0.2$	0.2
	Sum = 3.0			Numerator:	0.6

Semantic Score = 0.6 / 3.0 = 0.2

Severity Score Calculation: (Golden Severity = 5)

Rank	Weight (w_i)	Predicted Severity	Sev. Distance (D_i = 1 + |5 - pred|)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	mild (1)	5	$1.0 * (5-5)^2 = 0.0$	0.0
2	0.8	rare (5)	1	$0.8 * (5-1)^2 = 12.8$	12.8
3	0.6	severe (3)	3	$0.6 * (5-3)^2 = 2.4$	2.4
4	0.4	mild (1)	5	$0.4 * (5-5)^2 = 0.0$	0.0
5	0.2	critical (4)	2	$0.2 * (5-2)^2 = 1.8$	1.8
	Sum = 3.0			Numerator:	17.0

Severity Score = 17.0 / 3.0 = 5.67

5. Interpretation of Combined Scores

Presenting the Semantic and Severity scores side-by-side allows for a more holistic interpretation of the LLM's performance on individual cases.

Case 1 (Best Performance - Patient ID 31): This case demonstrated strong performance, scoring well on both Semantic (8.25) and Severity (13.0) metrics. The model identified the "Exact Synonym" at rank 1 and included relevant related diagnoses ("Exact/Broad Disease Group"). Severity matching was also strong, featuring perfect matches in the top ranks. It is noteworthy that the scores for this case were calculated using N=3 (sum of weights = 2.4), as only three diagnoses were provided in the LLM output. By normalizing the total weighted 'goodness' score by the sum of weights ((\sum w_i)), the final score reflects the average quality across the provided ranks. This normalization inherently penalizes longer lists that might include lower-quality items, as the denominator ((\sum w_i)) grows with list length (N) while the numerator might not increase proportionally if poor suggestions are added. Consequently, this case represents a high-quality differential diagnosis list according to these metrics.

Case 2 (Medium Performance - Patient ID 54):

This case achieved a good Semantic score (6.4) and a very good Severity score (14.4). Although the exact diagnosis was not listed, the top ranks were occupied by highly relevant "Broad Synonyms" and "Exact Disease Groups." Severity matching was excellent, with only a minor discrepancy at rank 4. This output represents a clinically useful list, offering closely related alternatives with appropriate severity alignment.

Case 3 (Worst Performance - Patient ID 20):

This case exhibited poor performance, scoring extremely low on Semantics (0.2) and poorly on Severity (5.67). The model failed to include the correct diagnosis ("Brugada syndrome") or any closely related synonyms within the top 5. The list predominantly consisted of unrelated conditions, and the severity matching was weak, showing significant

mismatches (e.g., mild/critical suggested vs. rare actual). This outcome signifies a low-quality list offering limited diagnostic value.

These examples demonstrate how the combination of Semantic and Severity scores provides a nuanced picture of the LLM's diagnostic reasoning capabilities that extends beyond simple accuracy. The normalization method, dividing by the sum of weights ((\sum w_i)), ensures the final score represents an average weighted quality. This approach effectively penalizes the inclusion of low-quality items in longer lists and facilitates fair comparisons between lists of potentially different lengths (N).

6. Advantages of Weighted Scoring

The Semantic and Severity scores offer significant advantages as complementary metrics to traditional Top-N accuracy for evaluating differential diagnoses. They provide a holistic evaluation by assessing the entire list, rather than solely focusing on the presence or absence of the correct diagnosis. Their rank awareness explicitly rewards models for placing more likely diagnoses higher in the list. Furthermore, they incorporate clinical relevance by considering semantic or severity "closeness," thereby differentiating between near misses and completely irrelevant suggestions. This results in a continuous score that allows for more nuanced comparisons between different models or prompting strategies. Ultimately, the emphasis on ranking and relevance aims for better alignment with clinical practice, reflecting how clinicians typically formulate and utilize differential diagnoses.

By employing these advanced metrics, Kernel29 facilitates a more robust and clinically meaningful assessment of LLM performance in the complex task of medical diagnosis. These scores allow researchers to move beyond simple accuracy measures and gain deeper insights into how well models capture the structural and clinical plausibility essential for a useful differential diagnosis.

7. Aggregating Scores for Overall Evaluation

While calculating scores for individual clinical cases provides valuable insights, evaluating the overall performance of a specific model-prompt combination across an entire dataset necessitates a method for score aggregation. Kernel29 employs a multi-step process involving score rescaling followed by weighted aggregation to achieve this.

7.1 Rescaling Individual Scores

As a preliminary step before aggregation, the individual Semantic and Severity scores, which initially range from 0 to a theoretical maximum of 16,calculated as:

$$w_1 \times (D_{max} - D_{min})^2 = 1.0 \times (5 - 1)^2 = 16$$

are rescaled to a standardized range of [-1, 1]. This normalization facilitates comparison across metrics and is a prerequisite for the subsequent weighted aggregation and Cartesian visualization. The rescaling transformation is defined as:

$$\text{score}_{\text{rescaled}} = \left(\frac{\text{score}_{\text{original}} \times 2.0}{\text{score}_{\text{max}}}\right) - 1.0$$

Where `score_original` is the calculated Semantic or Severity score (0-16) for a single case, and `score_max` is the theoretical maximum score (16.0). Under this transformation, an original score of 16.0 maps to +1.0 (best), 8.0 maps to 0.0 (neutral), and 0.0 maps to -1.0 (worst).

7.2 Weighted Aggregation

A simple arithmetic mean of rescaled scores might mask clinically significant variations in performance. A model could perform well on average yet occasionally generate severely misleading differential diagnoses (very low scores) on challenging cases. In clinical practice, such significant errors, which could lead to diagnostic delays or mismanagement, are often more critical than minor inaccuracies on straightforward cases. To better reflect this clinical reality and provide a measure more sensitive to potentially harmful outputs, Kernel29 employs a weighted aggregation strategy. This approach assigns higher weights to cases where the model performed poorly (lower rescaled scores), ensuring that instances of significant clinical misalignment disproportionately influence the final aggregate score. This yields a summary statistic that is more indicative of the model's reliability and robustness against critical failures.

The aggregation process involves two main steps:

1. Calculate Case Weight: A weight (W_{case}) is determined for each individual case based on its rescaled score using a reversed logistic sigmoid function. This function is chosen for its smooth transition properties. The general form, allowing for tunable sensitivity, is:

$$W_{\text{case}} = \frac{1.0}{1.0 + \exp\left(k \times (\text{score}_{\text{rescaled}} - x_0)\right)}$$

   - `score_rescaled`: The input rescaled score for the case (-1 to 1).
   - `k`: The steepness parameter. Higher values create a sharper transition in weight assignment around the midpoint.
   - `x_0`: The midpoint parameter, representing the rescaled score at which the weight equals 0.5. Shifting `x0` adjusts the threshold for what constitutes a "neutral" score in terms of weighting.
   - The function is effectively reversed (due to the positive exponent term `k \times ...` combined with the reciprocal structure) such that it assigns higher weights to scores below the midpoint `x0` and lower weights to scores above it.
   - Kernel29 Default: The standard implementation uses `x_0 = 0`, centering the neutral weight point (0.5) precisely at a rescaled score of 0.

2. Calculate Weighted Average: The final aggregated score for a specific model-prompt combination over the dataset is computed as the weighted average of the individual rescaled scores:
Score_aggregated = (Σ (W_case × score_rescaled)) / (Σ W_case)

This calculation yields a single aggregated, rescaled Semantic score and a single aggregated, rescaled Severity score (both ranging from -1 to 1) for each model-prompt combination evaluated.

7.2.1 Aggregation Examples under Different Difficulty Settings

To illustrate the impact of the weighting parameters (`k` and `x0`) on the final aggregated score under varying performance distributions, Table 4 analyzes three hypothetical scenarios using different weighting configurations, termed "difficulty settings":

- Easy Setting: `k=1, x0=0.3` (Gentle slope, midpoint shifted right – less emphasis on penalizing scores slightly below 0)
- Medium Setting: `k=2, x0=0` (Moderate slope, midpoint at zero – moderate penalty for scores below 0)
- Hard Setting: `k=3, x0=0` (Sharp slope, midpoint at zero – strong emphasis on penalizing scores below 0)

Scenario 1: Mixed Rescaled Scores (Simple Avg = -0.0625)

Easy Setting (k=1, x0=0.3) - Agg. Score: -0.289

Field	Case A1	Case A2	Case A3	Case A4	SUMS
Score (s)	+1.0	-0.5	+0.25	-1.0	
Weight Calc (Easy)	`1/(1+e^(1*(1-0.3)))`	`1/(1+e^(1*(-0.5-0.3)))`	`1/(1+e^(1*(0.25-0.3)))`	`1/(1+e^(1*(-1-0.3)))`	
Weight (Easy)	0.332	0.690	0.512	0.786	2.320
Contrib. (Easy)	0.332	-0.345	0.128	-0.786	-0.671

Medium Setting (k=2, x0=0) - Agg. Score: -0.490

Field	Case A1	Case A2	Case A3	Case A4	SUMS
Score (s)	+1.0	-0.5	+0.25	-1.0	
Weight Calc (Med)	`1/(1+e^(2*(1-0)))`	`1/(1+e^(2*(-0.5-0)))`	`1/(1+e^(2*(0.25-0)))`	`1/(1+e^(2*(-1-0)))`	
Weight (Med)	0.119	0.731	0.378	0.881	2.109
Contrib. (Med)	0.119	-0.366	0.094	-0.881	-1.034

Hard Setting (k=3, x0=0) - Agg. Score: -0.577

Field	Case A1	Case A2	Case A3	Case A4	SUMS
Score (s)	+1.0	-0.5	+0.25	-1.0	
Weight Calc (Hard)	`1/(1+e^(3*(1-0)))`	`1/(1+e^(3*(-0.5-0)))`	`1/(1+e^(3*(0.25-0)))`	`1/(1+e^(3*(-1-0)))`	
Weight (Hard)	0.047	0.818	0.321	0.952	2.138
Contrib. (Hard)	0.047	-0.409	0.080	-0.952	-1.234

Scenario 2: Mostly High Rescaled Scores (Simple Avg = +0.65)

Easy Setting (k=1, x0=0.3) - Agg. Score: +0.537

Field	Case B1	Case B2	Case B3	Case B4	SUMS
Score (s)	+1.0	+0.8	+0.9	-0.1	
Weight Calc (Easy)	`1/(1+e^(1*(1-0.3)))`	`1/(1+e^(1*(0.8-0.3)))`	`1/(1+e^(1*(0.9-0.3)))`	`1/(1+e^(1*(-0.1-0.3)))`	
Weight (Easy)	0.332	0.378	0.354	0.599	1.663
Contrib. (Easy)	0.332	0.302	0.319	-0.060	0.893

Medium Setting (k=2, x0=0) - Agg. Score: +0.333

Field	Case B1	Case B2	Case B3	Case B4	SUMS
Score (s)	+1.0	+0.8	+0.9	-0.1	
Weight Calc (Med)	`1/(1+e^(2*(1-0)))`	`1/(1+e^(2*(0.8-0)))`	`1/(1+e^(2*(0.9-0)))`	`1/(1+e^(2*(-0.1-0)))`	
Weight (Med)	0.119	0.168	0.142	0.550	0.979
Contrib. (Med)	0.119	0.134	0.128	-0.055	0.326

Hard Setting (k=3, x0=0) - Agg. Score: +0.147

Field	Case B1	Case B2	Case B3	Case B4	SUMS
Score (s)	+1.0	+0.8	+0.9	-0.1	
Weight Calc (Hard)	`1/(1+e^(3*(1-0)))`	`1/(1+e^(3*(0.8-0)))`	`1/(1+e^(3*(0.9-0)))`	`1/(1+e^(3*(-0.1-0)))`	
Weight (Hard)	0.047	0.083	0.063	0.574	0.767
Contrib. (Hard)	0.047	0.066	0.057	-0.057	0.113

Scenario 3: Mostly Low Rescaled Scores (Simple Avg = -0.65)

Easy Setting (k=1, x0=0.3) - Agg. Score: -0.738

Field	Case C1	Case C2	Case C3	Case C4	SUMS
Score (s)	-0.8	-0.9	-1.0	+0.1	
Weight Calc (Easy)	`1/(1+e^(1*(-0.8-0.3)))`	`1/(1+e^(1*(-0.9-0.3)))`	`1/(1+e^(1*(-1-0.3)))`	`1/(1+e^(1*(0.1-0.3)))`	
Weight (Easy)	0.750	0.768	0.786	0.450	2.754
Contrib. (Easy)	-0.600	-0.691	-0.786	0.045	-2.032

Medium Setting (k=2, x0=0) - Agg. Score: -0.753

Field	Case C1	Case C2	Case C3	Case C4	SUMS
Score (s)	-0.8	-0.9	-1.0	+0.1	
Weight Calc (Med)	`1/(1+e^(2*(-0.8-0)))`	`1/(1+e^(2*(-0.9-0)))`	`1/(1+e^(2*(-1-0)))`	`1/(1+e^(2*(0.1-0)))`	
Weight (Med)	0.832	0.858	0.881	0.450	3.021
Contrib. (Med)	-0.666	-0.772	-0.881	0.045	-2.274

Hard Setting (k=3, x0=0) - Agg. Score: -0.769

Field	Case C1	Case C2	Case C3	Case C4	SUMS
Score (s)	-0.8	-0.9	-1.0	+0.1	
Weight Calc (Hard)	`1/(1+e^(3*(-0.8-0)))`	`1/(1+e^(3*(-0.9-0)))`	`1/(1+e^(3*(-1-0)))`	`1/(1+e^(3*(0.1-0)))`	
Weight (Hard)	0.917	0.937	0.952	0.426	3.232
Contrib. (Hard)	-0.734	-0.843	-0.952	0.043	-2.486

The table demonstrates how weighted aggregation scores diverge from the simple arithmetic average. The choice of difficulty setting (`Easy`, `Medium`, `Hard`) reflects different tolerances for poor performance.

* Scenario 1 (Mixed): The simple average (-0.0625) masks the negative scores. The Easy setting (-0.289) begins to reflect this, while the Medium (-0.490) and Hard (-0.577) settings increasingly penalize the low scores (A2, A4), indicating higher sensitivity to potential clinical errors.

* Scenario 2 (Mostly High): The simple average (+0.65) is high. However, the Medium (+0.333) and Hard (+0.147) settings show a substantial reduction due to the single negative score (B4), highlighting how even infrequent poor performance can be flagged when using stricter aggregation settings.

* Scenario 3 (Mostly Low): All scores are negative, indicating poor performance. The weighted scores (Medium: -0.753, Hard: -0.769) are lower than the simple average (-0.65), further emphasizing the predominance of clinically problematic outputs.

Overall Conclusion on Aggregation: Weighted aggregation offers a tunable mechanism to summarize model performance across a dataset, allowing for different emphasis on performance in challenging cases (those resulting in lower scores). Compared to the simple arithmetic average, which treats all cases equally, weighted aggregation provides a more sophisticated measure that can be tuned to reflect the desired emphasis on performance in challenging cases.

8. Cartesian Coordinate Visualization

The aggregated, rescaled Semantic and Severity scores (−1 to +1) for each model-prompt combination provide a powerful means for comparative performance visualization using a 2D Cartesian plot.

- The X-axis represents the Aggregated Rescaled Severity Score.
- The Y-axis represents the Aggregated Rescaled Semantic Score.

Each point plotted corresponds to a specific model-prompt configuration, and its position within the four quadrants reveals its overall performance characteristics relative to both metrics:

- Top-Right Quadrant ($X > 0$, $Y > 0$): Indicates good performance in both Severity Matching and Semantic Relevance. Configurations falling in this quadrant are generally preferred, demonstrating strength on both evaluation dimensions. The ideal target is the top-right corner (+1, +1).
- Top-Left Quadrant ($X < 0$, $Y > 0$): Suggests poor Severity Matching despite good Semantic Relevance. Models here tend to identify clinically related diagnoses but struggle with aligning their severity levels with the case's actual severity.
- Bottom-Left Quadrant ($X < 0$, $Y < 0$): Represents poor performance on both Severity Matching and Semantic Relevance. Configurations in this area are generally considered the least desirable.

- Bottom-Right Quadrant (X > 0, Y < 0): Indicates good Severity Matching but poor Semantic Relevance. Models plotted here might suggest diagnoses with appropriate severity levels, but these diagnoses are often semantically distant from the correct one.

This visualization facilitates the immediate identification of high-performing model-prompt combinations (those closest to the top-right corner) and aids in understanding performance trade-offs or specific weaknesses (e.g., a model excelling in semantics but failing in severity alignment). It offers a concise, multi-dimensional overview of performance across an entire dataset, providing richer insights than single accuracy metrics alone.

9. Results Visualization and Interpretation

The quantitative metrics described—Semantic Relationship Score and Severity Matching Score—along with the aggregation methods, provide a foundation for visualizing and interpreting LLM performance in differential diagnosis tasks. This section presents example visualizations using results from the `c3opus` (Claude 3 Opus) and `llama2_7b` models evaluated on the `PUMCH_ADM` dataset (aliased as `ramedis`) with the `dxgpt_improved` prompt.

9.1 Distribution of Individual Case Scores

Before aggregating scores, examining the distribution of the raw (0-16) Semantic and Severity scores across all individual cases in the dataset can reveal valuable insights into a model's consistency and typical performance patterns, particularly on challenging clinical datasets like `PUMCH_ADM`. Violin plots combined with swarm plots (Figures 1 and 2) visualize these distributions.

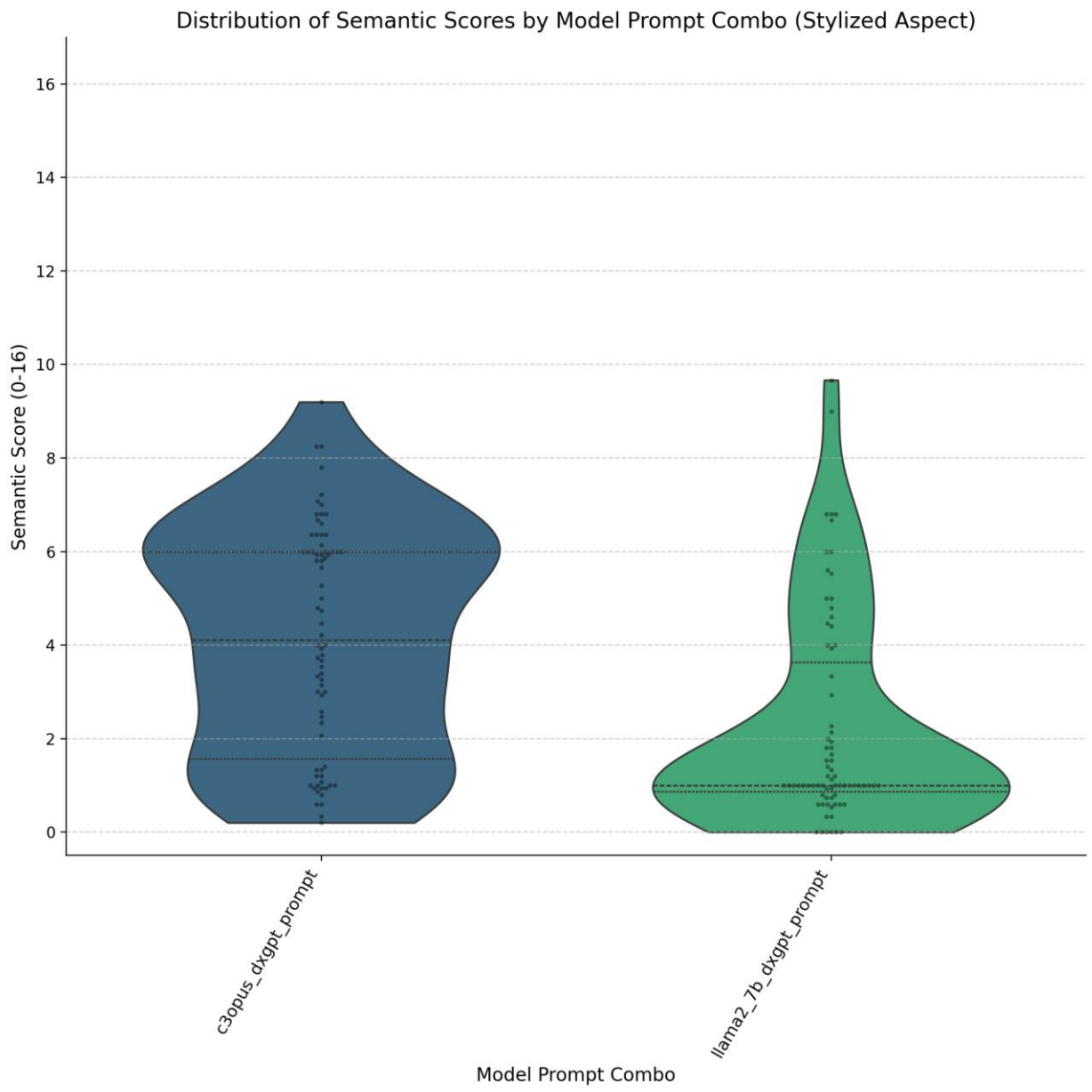

Figure 1: Distribution of Semantic Scores by Model-Prompt Combination. This plot shows the distribution of the raw Semantic Relationship Scores (0-10.33) for individual patient cases for `c3opus_dxgpt_improved` and `llama2_7b_dxgpt_improved`. The width of the violin indicates the density of cases at that score level.

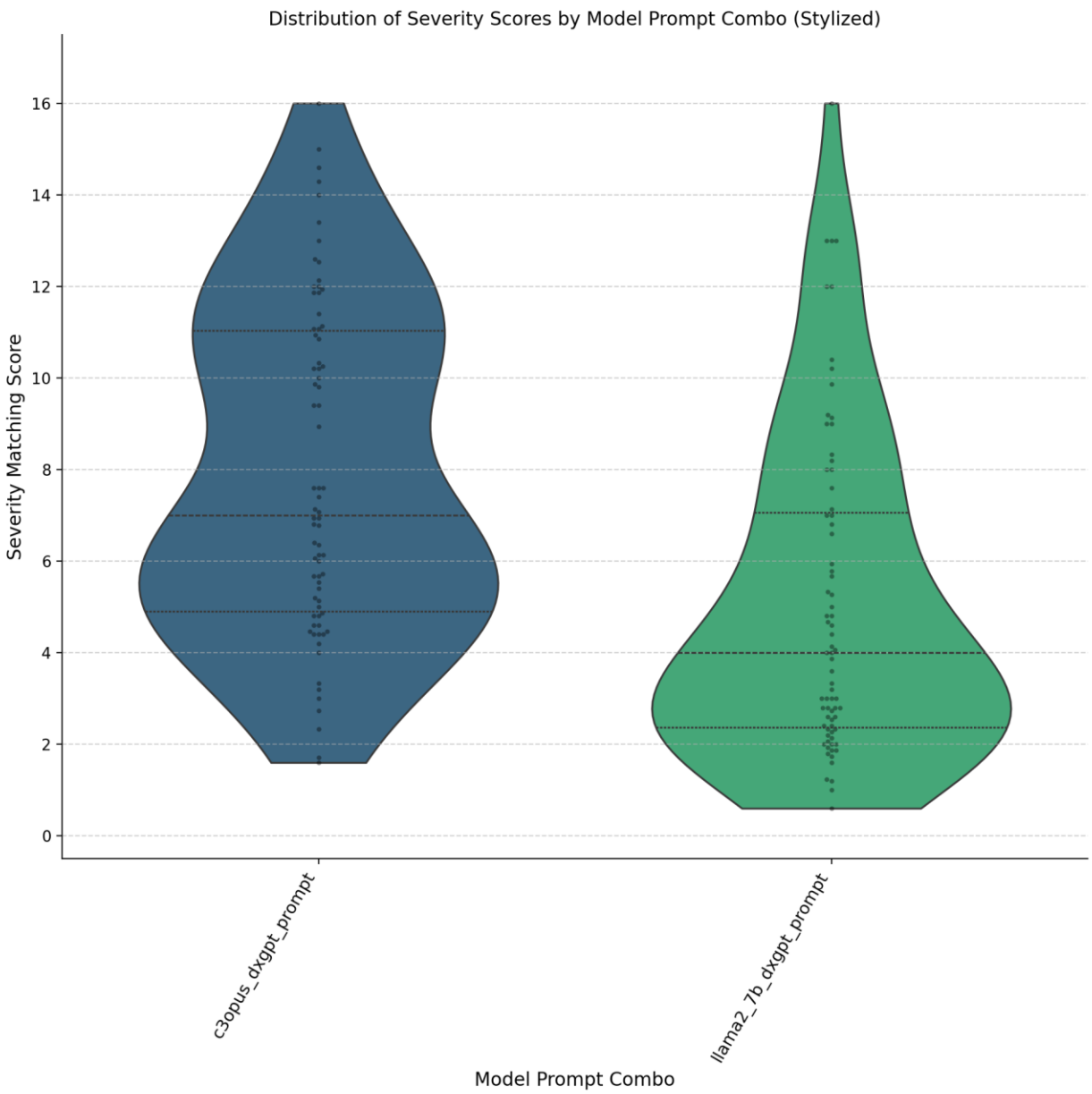

Figure 2: Distribution of Severity Matching Scores by Model-Prompt Combination. This plot displays the distribution of the raw Severity Matching Scores (0-16) for the same model-prompt combinations.

Observing these distributions provides a granular view beyond simple averages. Semantic Scores (Fig 1): The plot for `c3opus` shows a wide distribution, with a concentration of scores in the mid-range (around 4-8) but also a notable number of cases scoring lower (<4) and a smaller tail towards higher scores (>8). This suggests variable performance in semantic relevance; while often achieving moderate relevance, it struggles frequently on this dataset. The `llama2_7b` plot shows a distribution heavily skewed towards very low scores (0-2), with a long, thin tail extending upwards.

This indicates generally poor semantic performance, with only occasional instances of moderate or good relevance. The wider shape of the `c3opus` violin compared to the narrow base of `llama2_7b` suggests `c3opus` attempts relevance more broadly but less consistently achieves high scores than `llama2_7b` achieves low scores. Severity Scores (Fig 2): The `c3opus` severity scores also show a broad distribution, perhaps slightly more concentrated in the mid-to-high range (around 5-12) compared to its semantic scores, but still with significant spread. This implies moderate but inconsistent severity matching.

The `llama2_7b` severity scores are again heavily concentrated at lower values (2-6), with a sparser tail extending higher. This points towards frequent mismatches in severity assessment for `llama2_7b`. Overall: For both models, the wide spread and the presence of many low-scoring cases (especially visible in the swarm plots within the violins) underscore the difficulty of the dataset. Neither model consistently produces high-quality differential diagnoses according to these metrics across all cases. `c3opus` appears generally more capable than `llama2_7b` on both dimensions, but exhibits considerable variability.

9.2 Aggregated Performance Comparison

The Cartesian coordinate plot (Figure 3) summarizes the overall performance using weighted aggregation ("Hard" setting: `k=3, x0=0`). This setting strongly penalizes poor performance (negative rescaled scores), making the aggregated score sensitive to the presence of clinically significant errors observed in the distributions above.

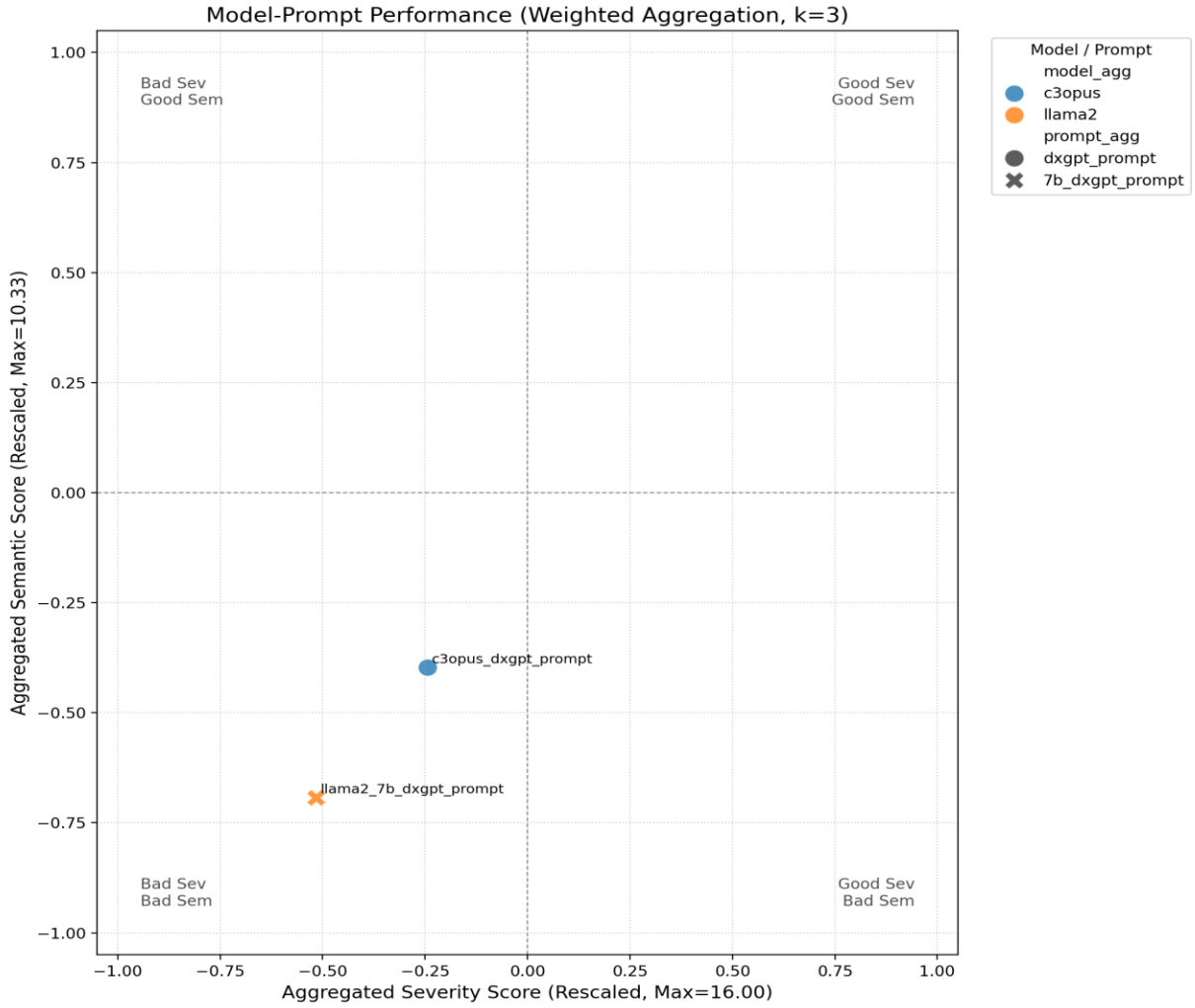

Figure 3: Aggregated Model-Prompt Performance Comparison. This plot shows the final aggregated Semantic and Severity scores for `c3opus_dxgpt_improved` and `llama2_7b_dxgpt_improved` using weighted aggregation (k=3, x0=0).

Figure 3 confirms the overall trend seen in the distributions. Both models land in the bottom-left quadrant ("Bad Sev, Bad Sem"), indicating negative aggregated scores on both axes. This outcome, especially under the "Hard" aggregation (`k=3`), is heavily influenced by the numerous low-scoring cases observed in Figures 1 and 2.

Even if a model performs well on some cases, the “Hard” setting ensures that frequent or severe clinical misalignments (low individual scores) pull the aggregate score down significantly, reflecting a lower overall clinical reliability. `c3opus_dxgpt_improved` (-0.38 Sem, -0.41 Sev, based on visual estimate) performs less poorly than `llama2_7b_dxgpt_improved` (-0.68 Sem, -0.61 Sev, based on visual estimate).

While relatively better, `c3opus`’s position still signifies substantial challenges with semantic relevance and severity matching on average, when weighted by difficulty. The aggregation highlights that, despite potential successes on individual cases (visible in Figures 4/5 later), the overall performance, particularly considering the penalty for errors, is suboptimal for both models on this challenging task. This emphasizes the need for improvements, potentially in the models themselves or the prompting strategies, to achieve reliable clinical utility.

9.3 Individual Case Performance Analysis

Beyond aggregated summaries, the same Cartesian coordinate system provides a powerful lens to visualize the performance on each individual case for a given model-prompt combination (Figures 4 and 5). This granular view, when linked back to the specific diagnoses, is crucial for understanding the nuances behind the aggregated scores and identifying patterns of diagnostic successes and failures across different types of clinical scenarios.

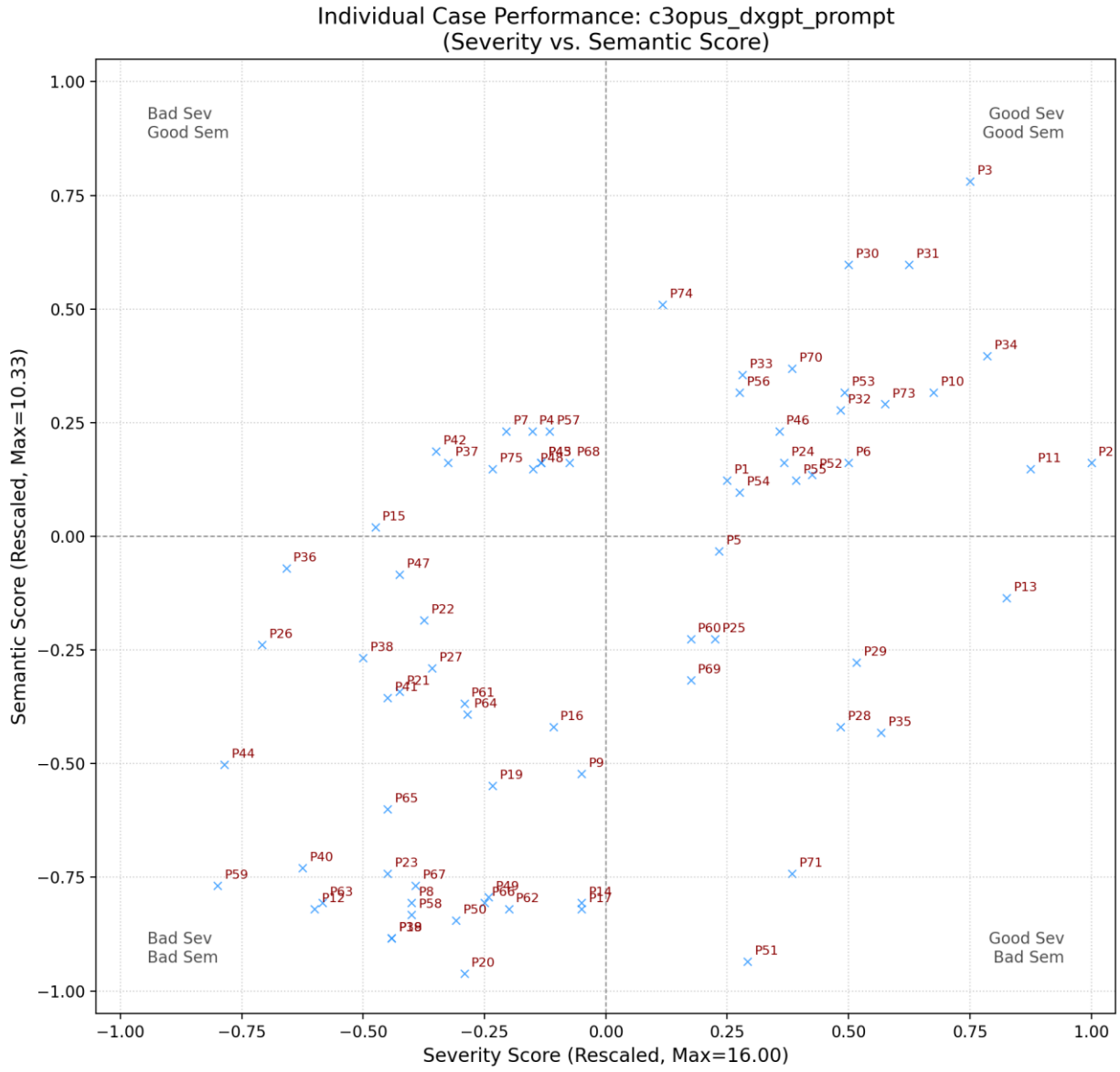

Figure 4: Individual Case Performance for `c3opus_dxgpt_prompt`. Each point represents a single patient case plotted according to its rescaled Semantic and Severity scores.

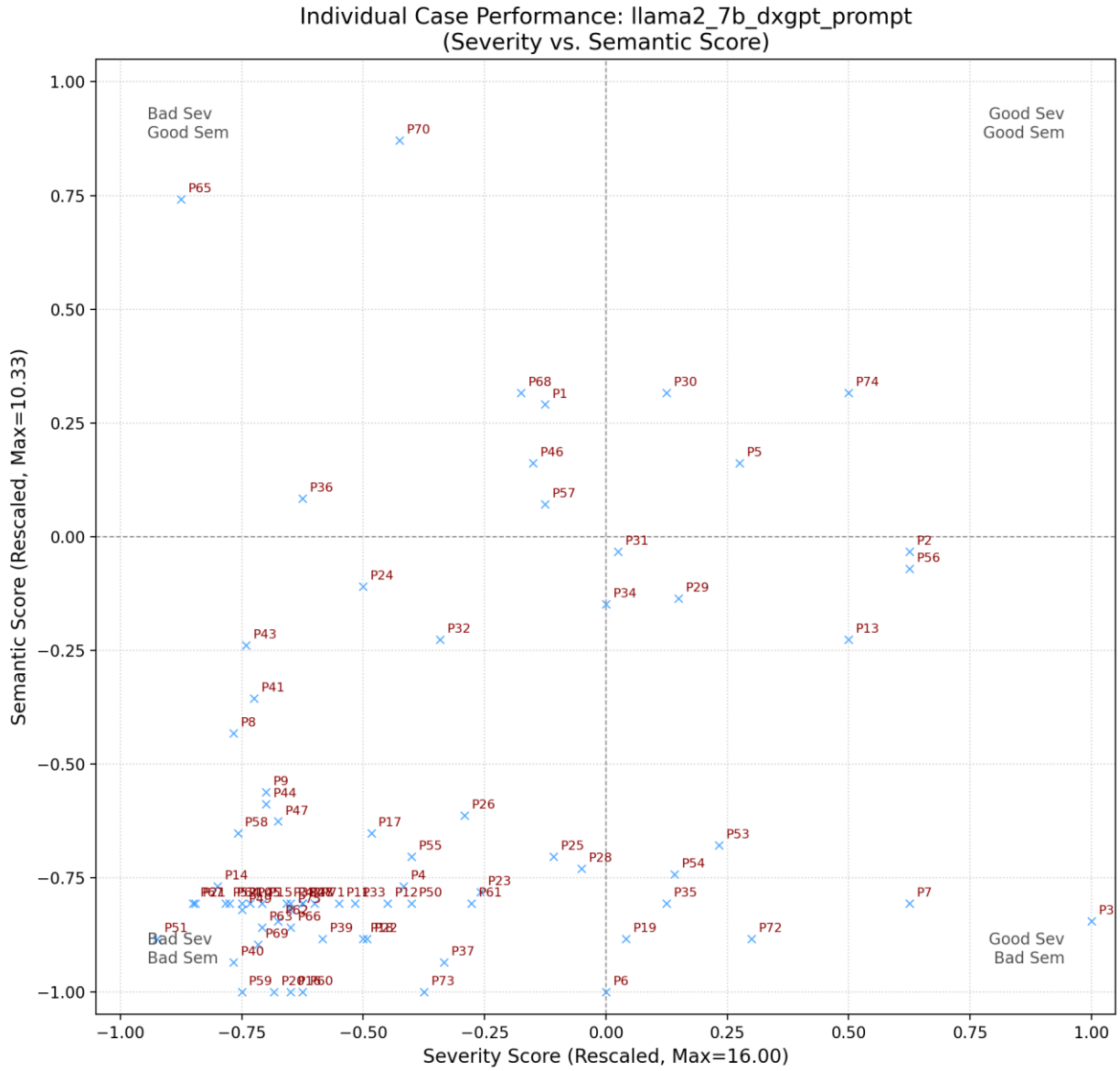

Figure 5: Individual Case Performance for `llama2_7b_dxgpt_prompt`. Similar to Figure 4, showing individual case performance for the llama2 model.

Figures 4 and 5 unpack the aggregated scores from Figure 3, revealing wide performance variation. Linking points to the underlying data allows for detailed error analysis:

* Identifying Successes: Cases in the top-right quadrant represent successful diagnoses. For example, Patient ID 3 (Golden: McCune-Albright syndrome, rare) was handled well by `c3opus`, which ranked the exact synonym first with correct severity. Patient ID 5 (also McCune-Albright) was similar, with `c3opus` ranking the exact synonym second but correctly identifying Acromegaly (Broad Disease Group, severe) first, leading to good scores. `llama2_7b` also successfully identified McCune-Albright as the exact synonym for

Patient ID 5 (though not for ID 3), placing this case in the top-right for that model. These represent relatively "easy" cases for the models.

* Pinpointing Failures: Cases in the bottom-left quadrant signify poor performance. For Patient ID 6 (Golden: McCune-Albright syndrome, rare), `llama2_7b` generated a list entirely composed of unrelated ophthalmological conditions (e.g., Retinitis Pigmentosa, Leber Congenital Amaurosis), resulting in extremely poor semantic and severity scores, placing it deep in the bottom-left. Similarly, for Patient ID 14 (Golden: Desminopathy, rare), `llama2_7b` ranked unrelated "Uterine leiomyoma" first, leading to very poor scores. These "hard" cases highlight significant diagnostic failures.

* Analyzing Error Types: The off-quadrants reveal specific weaknesses. Cases in the top-left (Good Semantics, Bad Severity) occur when the model identifies related conditions but misjudges severity. For Patient ID 8 (Golden: Neonatal Marfan syndrome, rare), `c3opus` suggested related severe heart conditions (Aortic/Mitral valve regurgitation), capturing the disease group but mismatching the severity. `llama2_7b` similarly suggested related cardiac issues for Patient ID 8 but with varying severities (mild Mitral valve prolapse, severe Aortic stenosis, critical Endocarditis), also leading to poor severity matching despite reasonable semantic grouping. Such errors could lead to clinical mismanagement. Cases in the bottom-right (Bad Semantics, Good Severity) are rarer. One might hypothetically see this if, for a rare cardiac condition, the model suggested an unrelated rare neurological condition – the severity matches ("rare"), but the suggestion is diagnostically incorrect.

* Explaining Aggregated Scores: The overall pattern explains the aggregated scores (Figure 3). The numerous successes (top-right) for `c3opus` are counterbalanced by its failures (bottom-left and off-quadrants), leading to a negative but less extreme aggregated score. For `llama2_7b`, the heavy concentration of points in the bottom-left (e.g., P6, P14) and top-left (e.g., P8) quadrants, indicating frequent semantic failures or severity mismatches, drives its strongly negative aggregated scores under the "Hard" weighting. * Clinical Utility Assessment: This case-level analysis, linking plot positions to actual diagnoses, allows researchers to identify specific clinical areas (e.g., rare syndromes, specific organ systems) where models excel or struggle. It enables targeted error analysis (e.g., "Does the model consistently underestimate severity for cardiac conditions?") vital for assessing clinical safety and guiding improvements.

This multi-level visualization approach—from score distributions to aggregated plots to detailed individual case breakdowns—provides a comprehensive framework for evaluating the clinical performance characteristics of LLMs.

Appendix: Metodological remarks

It is important to note that the semantic relationships and severity levels used in these calculations (including the example above) were derived using Llama 3.1 70B configured as an 'LLM-as-judge' for multi-class classification. This specific model was chosen primarily for efficiency and cost-effectiveness during this evaluation phase. For real-world clinical evaluation scenarios requiring higher robustness and accuracy, utilizing more advanced models (such as Gemini) would be advisable.

The prompts used to instruct Llama 3.1 70B as a classifier underwent four iterations for refinement. However, the classification results suggest that further improvements may still be necessary. For instance, in the analysis of Patient ID 3 (Golden: "McCune-Albright syndrome", Severity: rare), the predicted diagnosis at Rank 4, "Endocrine disorders" (Severity: moderate), was classified as a "Broad Synonym" by the LLM-as-judge.

- Golden Diagnosis: "McCune-Albright syndrome" (Severity: rare = 5)
- LLM Output:

Rank	Predicted Diagnosis	Severity	Semantic Relationship
1	McCune-Albright syndrome	rare	Exact Synonym
2	Neurofibromatosis type 1	rare	Broad Disease Group
3	Polyostotic fibrous dysplasia	rare	Broad Synonym
4	Endocrine disorders	moderate	Broad Synonym
5	Skeletal dysplasias	rare	Broad Synonym

- Semantic Score Calculation:

Rank	Weight (w_i)	Semantic Relationship	Sem. Distance (D_i)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	Exact Synonym	1	$1.0 * (5-1)^2 = 16.0$	16.0
2	0.8	Broad Disease Group	4	$0.8 * (5-4)^2 = 0.8$	0.8
3	0.6	Broad Synonym	2	$0.6 * (5-2)^2 = 5.4$	5.4
4	0.4	Broad Synonym	2	$0.4 * (5-2)^2 = 3.6$	3.6
5	0.2	Broad Synonym	2	$0.2 * (5-2)^2 = 1.8$	1.8
	Sum = 3.0			Numerator:	27.6

Semantic Score = 27.6 / 3.0 = 9.2

- Severity Score Calculation: (Golden Severity = 5)

Rank	Weight (w_i)	Predicted Severity	Sev. Distance ($D_i = 1 + \|5 - pred\|$)	Score Calc: $w_i*(5-D_i)^2$	Contribution
1	1.0	rare (5)	1	$1.0 * (5-1)^2 = 16.0$	16.0
2	0.8	rare (5)	1	$0.8 * (5-1)^2 = 12.8$	12.8
3	0.6	rare (5)	1	$0.6 * (5-1)^2 = 9.6$	9.6
4	0.4	moderate (2)	4	$0.4 * (5-4)^2 = 0.4$	0.4
5	0.2	rare (5)	1	$0.2 * (5-1)^2 = 3.2$	3.2
	Sum = 3.0			Numerator:	42.0

Severity Score = 42.0 / 3.0 = 14.0

While McCune-Albright syndrome involves endocrine dysfunction, classifying the very general term “Endocrine disorders” as a “Broad Synonym” might be considered overly permissive and lack the specificity desired for rigorous clinical evaluation. Then, the differential diagnosis is classified as very good by the semantic score while not true. This highlights a potential area for refinement, either through more sophisticated prompting techniques or by employing a more capable judge model.

Consequently, the results presented using this Llama 3.1 70B judge should be interpreted considering it as an initial implementation approach, primarily intended for demonstrating the scoring metrics rather than representing a definitive, clinically validated classification.

CHAPTER 4: The Kernel29 Framework Architecture

Framework Overview

Previous chapters detailed the significant hurdles in large-scale LLM experimentation: managing combinatorial complexity and ad-hoc engineering practices(Chapter 1, Tables 1-4), ensuring robust data standardization (Chapter 2, `bat29`), and implementing meaningful evaluation metrics (Chapter 3). The sheer scale and intricacy demand a departure from unstructured approaches, which inevitably lead to the inefficiencies, irreproducibility, and scalability issues outlined previously.

Kernel29 provides this essential architectural solution. It offers a structured environment specifically designed to tackle the interwoven engineering and data science demands of systematic LLM research and development. By establishing clear conventions and providing integrated core components, Kernel29 enables maintainable, reproducible, and scalable workflows. The framework is principally composed of:

1. A standardized module architecture for task-specific code.
2. The `lapin` library for unified LLM interaction, configuration, and execution.
3. A central `db` component for structured, persistent storage of all experimental data, including inputs, configurations, LLM outputs, and evaluations.

Core Components: `lapin` and `db`

Addressing the complexity and data management needs of large-scale LLM experimentation requires dedicated tooling. Kernel29 provides two foundational components for this purpose: `lapin` for managing LLM interactions and experimental orchestration, and `db` for ensuring data persistence and integrity.

`Lapin`: LLM Interaction and Orchestration

Extensive LLM experimentation, demands a centralized solution capable of managing scale and complexity effectively. Otherwise, operational challenges (e.g. handle heterogeneous LLM API protocols) would dominate development effort, hindering research progress.

`lapin` serves as this crucial infrastructural component within Kernel29. It encapsulates the necessary logic for handling API diversity, managing configurations via aliases, building prompts dynamically, and orchestrating asynchronous execution. By providing this standardized toolkit, `lapin` abstracts away operational challengues, enabling researchers and developers to focus on designing experiments, analyzing results, and advancing the core scientific objectives of the project.

Key Features of `lapin`:

Feature	Description	Relevant `lapin` Modules/Classes
API Abstraction	Provides unified handlers to interact with different LLM	`lapin.handlers` (e.g., `AsyncModelHandler`)

Feature	Description	Relevant `lapin` Modules/Classes
	provider APIs, masking underlying differences.	
Configuration	Defines and manages LLM configurations (model details, API keys, parameters) through dedicated objects and loading mechanisms.	`lapin.core.config.LapinConfig`, `load_config`
Component Registry	Allows `LapinConfig` and `PromptBuilder` instances to be registered with unique string aliases for easy reference and loading.	`@register_config`, `@register_prompt` decorators
Dynamic Loading	Enables runtime selection and instantiation of registered configurations and prompt builders based on their aliases, typically via CLI args.	Implicitly used by runner scripts via registration
Prompt Engineering	Provides a structured way (`PromptBuilder`) to define, manage, and version different prompt strategies programmatically.	`lapin.prompt_builder.base.PromptBuilder`
Batch Processing	Offers utilities for efficient, asynchronous batching of API requests to improve performance when processing large datasets.	`lapin.utils.async_batch.process_all_batches`

db: Data Persistence and Integrity

A core principle of Kernel29 is a data-centric workflow, shifting from fragile file-based I/O to a structured, persistent database layer (`db`). This approach is crucial for ensuring the reproducibility, traceability, and analytical potential of experimental results. By centralizing inputs, outputs, configurations, and evaluations in a database (typically using SQLAlchemy), Kernel29 facilitates robust data management.

Key Database Tables (Illustrative, based on Model Definitions):

The database is organized into schemas (like `bench29`, `registry`, `llm`, `prompts`). Below are some important tables, primarily from the `bench29` schema, ordered by likely importance for core experimental workflows:

SQLAlchemy Model Class	Description	Schema
`CasesBench`	Stores the primary clinical case data used as input for LLM tasks, including the original text and source metadata (e.g., hospital).	`bench29`
`CasesBenchDiagnosis`	Stores the gold standard (correct) diagnosis associated with each case in `CasesBench`, used for evaluation.	`bench29`
`LlmDifferentialDiagnosis`	Records the raw output generated by a specific LLM (`Models`) using a specific prompt (`Prompt`) for a given case (`CasesBench`).	`bench29`
`DifferentialDiagnosis2Rank`	Parses the raw `LlmDifferentialDiagnosis` output into a ranked list of predicted diagnoses, storing each diagnosis, its rank, and reasoning.	`bench29`
`DifferentialDiagnosis2Severity`	Stores severity judgments (linking to `SeverityLevels`) for each ranked diagnosis generated by an LLM or a judge.	`bench29`
`DifferentialDiagnosis2SemanticRelationship`	Stores semantic relationship judgments vs gold (linking to `DiagnosisSemanticRelationship`) for each ranked diagnosis.	`bench29`
`LlmAnalysis`	An aggregation table combining case, LLM output, ranking, and	`bench29`

SQLAlchemy Model Class	Description	Schema
	judgment information for comprehensive analysis.	
`CasesBenchMetadata`	Stores supplementary metadata about clinical cases, such as predicted complexity, specialty, or severity (links to `registry` tables).	`bench29`
`Models`	Defines the LLM models used in experiments, including their registered `alias`, full `name`, and `provider`.	`llm`
`Prompt`	Defines the different prompt templates used, identified by `alias`, storing `content`, metadata, and linking to the creator (`User`).	`prompts`
`SeverityLevels`	Registry table defining possible severity levels (e.g., Low, Medium, High) used for case metadata and diagnosis judgments.	`registry`
`DiagnosisSemanticRelationship`	Registry table defining categories for comparing predicted vs gold diagnoses (e.g., Exact Match, Parent, Child).	`registry`
`ComplexityLevels`	Registry table defining possible complexity levels for clinical cases.	`registry`

Note: This table list is illustrative and focuses on core experimental data. Additional tables like `PromptArguments`, `PromptRelationship`, etc., exist for more detailed tracking within the `prompts` schema.

Standardized Module Architecture

Building upon `lapin` and `db`, Kernel29 enforces a standardized structure for implementing specific LLM-driven tasks. Each task is encapsulated within its own module (e.g., `src/dxGPT` for diagnosis generation, `src/bench29` for evaluation). This modularity promotes separation of concerns, code reuse, and maintainability.

A standard module typically contains the following subdirectories, file types and functions:

Standard Directories:

Directory	Purpose
`models/`	Contains `LapinConfig` subclass definitions specific to the module, registered with `lapin` via aliases.
`prompts/`	Contains `PromptBuilder` implementations specific to the task, also registered via aliases.
`parsers/`	Includes functions responsible for parsing and structuring the raw outputs received from LLMs.
`queries/`	Intended for complex, task-specific database interactions or refined query logic not suitable for the central query repository (`src/db/queries/`). *Note: Query logic may also be found directly within core logic files.*
`serialization/`	(Optional) May contain Pydantic models for data validation and clear data contracts.

Standard File Patterns:

File Pattern / Type	Purpose	Example Filename(s)
`run_*_async.py`	The executable entry point for the module (Runner Script). Handles CLI arguments, setup, and orchestration via the Core Logic File.	`run_dxGPT_async.py`
`*_async.py`	Contains the main asynchronous functions (Core Logic) implementing the task's workflow (data fetching, prep, processing, storage). May currently contain embedded database query logic.	`dxGPT_async.py`
`*_models.py` inside `models/`	Defines specific `LapinConfig` classes, typically inheriting from `lapin` base configs, to specify model parameters, endpoints, etc.	`dxGPT_models.py`
`*_prompts.py` inside `prompts/`	Defines specific `PromptBuilder` classes, inheriting from `lapin.prompt_builder.base.PromptBuilder`, registered with `lapin`.	`dxGPT_prompts.py`
`*_parsers.py` inside `parsers/`	Defines parsing functions responsible for extracting structured data from raw LLM text responses.	`dxGPT_parsers.py`
`*_queries.py` inside `queries/`	Defines complex/specific database query functions for the module, potentially involving multiple tables or intricate logic beyond simple CRUD.	`dxGPT_queries.py`
`src/db/queries/{get or post}/*.py`	**Intended** location for reusable, general-purpose query functions (`get_*`, `add_*`) interacting with specific schemas, often grouped by operation type.	(e.g., `get/cases.py`)
`src/db/db_queries_{schema}.py`	**Current/Legacy** location where many general query functions currently reside due to development history, often grouping queries by schema (e.g., bench29).	`db_queries_bench29.py`

Common Function/Mode Conventions:

Convention Name / Pattern	Purpose	Example Functions/Modes
`main` / `main_async`	Main orchestrator function within the Core Logic file, called by the Runner Script.	`main`, `main_async`
`set_settings`	Initializes common resources like DB session, `lapin` handler, and prompt builder based on configuration/arguments.	`set_settings`
`retrieve_and_make_prompts`	Fetches input data from the database (using query functions, wherever they reside), processes/transforms it, and formats it for batch processing via `lapin`.	`retrieve_and_make_prompts`
`process_results`	Takes the raw output from `lapin`'s batch processing, parses it using functions from `parsers/`, and prepares results for database storage (using query functions, wherever they reside).	`process_results`
`get_*`	Functions responsible for retrieving data from the database. **Ideally centralized** in `src/db/queries/get/`, but **currently scattered** across various locations including legacy files (`src/db/db_queries_*.py`), the central query directory (`src/db/queries/get/`), module-specific query directories (`{module_root}/queries/`), and potentially embedded within core logic files or other utility	`get_cases`, `get_model_id`

Convention Name / Pattern	Purpose	Example Functions/Modes
	directories.	
`add_*`	Functions responsible for inserting or updating data in the database. **Ideally centralized** in `src/db/queries/post/`, but **currently scattered** across various locations including legacy files (`src/db/db_queries_*.py`), the central query directory (`src/db/queries/post/`), module-specific query directories (`{module_root}/queries/`), and potentially embedded within core logic files or other utility directories.	`add_semantic_results_to_db`, `add_differential_diagnoses`
`parse_*` (in `parsers/`)	Module-specific functions dedicated to extracting structured information from LLM text responses.	`parse_judged_semantic`, `parse_top5_xml`
Endpoint Mode vs. Debug Mode	Scripts often support execution via CLI arguments (Endpoint) for automation, or using hardcoded values (Debug) for development/testing.	Determined by presence of specific CLI args

This consistent architecture ensures that different tasks within the Kernel29 project share a common structure, simplifying navigation, understanding, and extension of the codebase while leveraging the core `lapin` and `db` functionalities.

CHAPTER 5: Generating and Evaluating LLM Diagnostics within the Kernel29 Framework

Introduction

Previous chapters established the necessity of standardized data processing (`bat29`, Chapter 2) and a robust architectural framework (`Kernel29`, `lapin`, `db`, Chapter 4) for managing the inherent complexities of large-scale Large Language Model (LLM) experimentation. This chapter details the practical application of this framework to the central task of generating differential diagnoses from clinical cases and subsequently evaluating the quality of these diagnoses using LLM-as-judge methodologies.

Leveraging the standardized clinical cases prepared by `bat29` and stored in the central database (`CasesBench` table), the Kernel29 framework orchestrates a multi-stage workflow. Initially, a target LLM processes the clinical cases to generate ranked lists of potential diagnoses along with supporting reasoning (Stage 1: `dxGPT`). Following generation, the diagnostic outputs undergo evaluation. First, a dedicated LLM "judge" assesses the semantic relationship (e.g., synonym, related, unrelated) between each LLM-generated diagnosis and the corresponding gold-standard diagnosis (Stage 2: Semantic Evaluation). Second, another LLM "judge" evaluates the clinical severity (e.g., mild, severe, critical) attributed to each generated diagnosis (Stage 3: Severity Evaluation). This chapter elucidates the purpose, technical design, and core logic of the software modules responsible for each stage, emphasizing the consistent architectural patterns employed and the integral roles of the `lapin` orchestration library and the `db` data persistence component.

General Technical Architecture: A Modular and Standardized Workflow

The diagnostic generation and evaluation process utilizes three distinct logical stages, implemented as software modules typically residing within the `src/` directory: `dxGPT` for initial diagnosis generation, and components within a `bench29` module for the evaluation stages (Semantic and Severity Judgment). Although functionally distinct, these modules adhere rigorously to the standardized Kernel29 architectural template detailed in Chapter 4.

This workflow heavily leverages the **`lapin` library** (introduced in Chapter 4) for orchestrating LLM interactions. `lapin` handles configuration management, unified model interaction via `AsyncModelHandler`, asynchronous batch processing using `process_all_batches`, and systematic prompt management via the `PromptBuilder` pattern. This allows different models (e.g., `--model llama3_70b`) and prompt strategies (e.g., `--prompt_alias dxgpt_standard`) to be selected at runtime via command-line arguments.

The stages follow the standard operational pattern established in Chapter 4: a command-line runner script (`run_*_async.py`) invokes the core asynchronous logic in a `*_async.py` file. This core module typically contains:

1. **`set_settings`**: Initializes the DB session, `lapin` handler, and prompt builder.
2. **`retrieve_and_make_prompts`**: Fetches and transforms data, uses the `PromptBuilder` to create prompts, and prepares wrapper objects for batching. The specific data sources and transformations vary significantly by stage.
3. **`process_results`**: Parses the raw LLM outputs from `lapin` and structures the data for storage, using stage-specific parsing logic.
4. **Database Storage**: Uses query functions to persist the structured results.

While the overall structure relies on `lapin` and the `db` component, the implementation details within `retrieve_and_make_prompts` and `process_results` are tailored to the specific task of each stage.

Stage 1: Differential Diagnosis Generation (`dxGPT`)

Purpose

The `dxGPT` module serves as the initial stage, responsible for generating differential diagnoses based on clinical case descriptions. It processes standardized clinical cases fetched from the database (`CasesBench` table), utilizing a user-specified LLM (e.g., `llama3_70b`, `gpt4o`, selected via the `--model` argument) and a chosen prompt strategy (e.g., `dxgpt_standard`, `dxgpt_json`, selected via `--prompt_alias`). The output consists of a ranked list of potential diagnoses, often accompanied by justifications comparing case features to diagnostic criteria, which is then stored back into the database.

Detailed Technical Implementation

Following the standard pattern, the `dxGPT` process is initiated by `run_dxGPT_async.py`, which calls the core logic in `dxGPT_async.py`. The key steps involved are:

1. **Setup (`set_settings`)**: Initializes the DB session, the `lapin` `AsyncModelHandler` for the target LLM (`--model`), and the `dxGPT` `PromptBuilder` (`--prompt_alias`).
2. **Input Preparation (`retrieve_and_make_prompts`)**: This function fetches specified `CasesBench` records, uses the loaded `PromptBuilder` to format the clinical narrative for each case according to the chosen strategy, and creates `DxGPTInputWrapper` objects containing the prompt text and necessary metadata (`case_id`, `model_id`, etc.) for `lapin`.
3. **Asynchronous Processing (`process_all_batches`)**: `lapin` manages the asynchronous dispatch of the wrapped prompts to the configured LLM API, returning a list of results.
4. **Result Processing (`process_results`)**: This function iterates through the `lapin` results. For each, it extracts the raw LLM response and metadata, selects the appropriate parser (based on `prompt_alias`, often from a `dxGPT/parsers` module) to structure the ranked diagnoses, and calls a database function (e.g., `insert_differential_diagnoses` from a `dxGPT` query module) to store the raw output (e.g., in `LlmDifferentialDiagnosis`) and the parsed rankings (e.g., in `DifferentialDiagnosis2Rank`).

Key internal software dependencies for this stage include the specific model configurations, prompt builders, parsers, and query functions defined within the `dxGPT` module. The primary database tables involved are `CasesBench` and `Models` (for reading input and model details) and `LlmDifferentialDiagnosis` and `DifferentialDiagnosis2Rank` (for writing the generated outputs).

Stage 2: Semantic Relationship Evaluation

Purpose

Following the generation and storage of differential diagnoses (in the `DifferentialDiagnosis2Rank` table), the subsequent stage focuses on evaluating the semantic relevance of each proposed diagnosis relative to the established gold-standard diagnosis for the case. This evaluation is performed by a dedicated script (e.g., `run_judge_semantic_async.py`), executing core logic typically found in a corresponding asynchronous module (e.g., `judge_semantic_async.py` within a `bench29` or similar evaluation module). It employs a designated LLM, configured as a "semantic judge", to classify the relationship (e.g., Exact Synonym, Broader Term, Related Condition, Unrelated) between the LLM-generated diagnosis and the gold standard.

Detailed Technical Implementation

The semantic evaluation process follows the standard architectural pattern, initiated via `run_judge_semantic_async.py`, but with distinct logic in data preparation and result processing.

1. **Execution**: The runner script takes arguments like the semantic judge model (`--semantic_judge`), the generation model being evaluated (`--differential_diagnosis_model`), and dataset identifiers (`--test_name`).
2. **Setup (`set_settings`)**: Initializes the DB session, the `AsyncModelHandler` for the specified semantic judge LLM, and the appropriate semantic `PromptBuilder`.
3. **Input Preparation (`retrieve_and_make_prompts`)**: **This function's logic is tailored to the semantic comparison task.**
   - Fetches relevant `DifferentialDiagnosis2Rank` records and the corresponding gold-standard diagnoses using database queries.
   - **Groups** the flat `DifferentialDiagnosis2Rank` records into nested dictionaries per case/model (e.g., using `create_nested_diagnosis_dict`).
   - **Formats** the data for the semantic judge: Transforms each grouped diagnosis set into a single text prompt containing *both* the gold-standard diagnosis *and* the target LLM's ranked list. This combined text is mapped to a composite key (e.g., `f"{case_id}_{model_id}_{diff_diag_id}"`) in a flat dictionary (e.g., using `dif_diagnosis_dict2plain_text_dict_with_real_diagnosis`).
   - **Creates an intermediate mapping**: Generates a dictionary mapping the composite keys back to original rank details (e.g., using `nested_dict2rank_dict`), crucial for correlating results later.

- **Wraps** the flat dictionary of prompts into `DiagnosisTextWrapper` objects for `lapin`.
- **Returns** the list of wrapper objects and the intermediate mapping dictionary.

4. **Asynchronous Processing (`process_all_batches`)**: `lapin` sends the prompts (containing gold standard + ranked list) to the semantic judge LLM.
5. **Result Processing (`process_results`)**: This function receives the raw results and the intermediate mapping dictionary.
   - For each result, it calls a specific parser (e.g., `parse_judged_semantic`) which uses the mapping dictionary to extract the semantic relationship category for *each* ranked diagnosis compared in the prompt.
   - Separates successful parses from failures.
   - **Returns** the list of successful parses and failures.
6. **Database Storage**: Calls a specific database function (e.g., `add_semantic_results_to_db`) to insert the successfully parsed judgments into the target table (e.g., `DifferentialDiagnosis2SemanticRelationship`).

Internal dependencies involve prompt builders, parsers, and utility functions within the evaluation module (`bench29`), common and specific query functions within the `db` component, and core `lapin` and `db.utils` components. Database interaction involves reading from `DifferentialDiagnosis2Rank`, `CasesBenchDiagnosis`, `CasesBench`, and `Models`, and writing the evaluation results to `DifferentialDiagnosis2SemanticRelationship`.

Stage 3: Severity Evaluation

Purpose

As a complementary evaluation, this stage assesses the clinical severity associated with each *individual* diagnosis proposed by the initial diagnosis-generating LLM. This helps understand if the LLM suggests appropriately severe conditions. The process is managed by a script (e.g., `run_judge_severity_async.py`), executing core logic in a corresponding asynchronous module (e.g., `judge_severity_async.py` within `bench29`). It uses a designated LLM configured as a "severity judge".

Detailed Technical Implementation

The severity evaluation follows the standard pattern initiated by `run_judge_severity_async.py`, differing from the semantic stage primarily in prompt content and parsing.

1. **Execution**: The runner script takes arguments like the severity judge model (`--severity_judge`), the model being evaluated (`--differential_diagnosis_model`), and dataset identifiers (`--test_name`).
2. **Setup (`set_settings`)**: Initializes the DB session, the `AsyncModelHandler` for the specified severity judge LLM, and the appropriate severity `PromptBuilder`.

3. **Input Preparation (`retrieve_and_make_prompts`): Logic focuses on preparing individual diagnoses for severity assessment.**
   - Fetches relevant `DifferentialDiagnosis2Rank` records.
   - **Groups** records into nested dictionaries per case/model (e.g., using `create_nested_diagnosis_dict`).
   - **Formats** data for the severity judge: Transforms each nested dictionary into a text string containing *only* the target LLM's ranked diagnosis list (gold standard is *not* included). Maps this text to a composite key in a flat dictionary (e.g., using `dif_diagnosis_dict2plain_text_dict`).
   - **Creates the intermediate mapping**: Generates a dictionary mapping composite keys back to original rank details (e.g., using `nested_dict2rank_dict`).
   - **Wraps** the flat dictionary into `DiagnosisTextWrapper` objects for `lapin`.
   - **Returns** the list of wrapper objects and the intermediate mapping dictionary.
4. **Asynchronous Processing (`process_all_batches`)**: `lapin` sends the prompts (containing only the ranked diagnoses list) to the severity judge LLM.
5. **Result Processing (`process_results`)**: Receives raw results and the intermediate mapping dictionary.
   - For each result, calls a specific severity parser (e.g., `parse_judged_severity`) which uses the mapping dictionary to extract the severity level for each diagnosis.
   - Separates successful parses from failures.
   - **Returns** the list of successful parses and failures.
6. **Database Storage**: Calls a specific database function (e.g., `add_severity_results_to_db`) to insert the successfully parsed severity judgments into the target table (e.g., `DifferentialDiagnosis2Severity`).

Internal dependencies involve prompt builders, parsers, and utility functions within the evaluation module (`bench29`), common and specific query functions within the `db` component, and core `lapin` and `db.utils` components. Database access includes reading `DifferentialDiagnosis2Rank`, `CasesBench`, and `Models`, and writing the evaluation results to `DifferentialDiagnosis2Severity`.

Conclusion

This chapter illustrated the application of the Kernel29 framework to orchestrate a sophisticated, multi-stage workflow for generating and evaluating LLM-based differential diagnoses. By adhering to the standardized module structures and leveraging the `lapin` library for LLM interaction and the `db` component for data persistence (as detailed in Chapter 4), the framework facilitates systematic and scalable experimentation. Each stage (`dxGPT`, Semantic Judgment, Severity Judgment) follows the established architectural pattern.

While maintaining structural consistency, the framework accommodates the unique requirements of each stage through tailored implementations within the `retrieve_and_make_prompts` and `process_results` functions, alongside stage-specific prompt builders and parsers. This highlights the framework's flexibility, demonstrated by the differing logic for prompt preparation (e.g., including gold standard for semantics vs. not for severity) and result parsing.

This modular, data-centric architecture enables robust assessment of LLM performance in clinical diagnosis. The structured capture of results in the database (`DifferentialDiagnosis2Rank`, `DifferentialDiagnosis2SemanticRelationship`, `DifferentialDiagnosis2Severity` tables) provides a foundation for the quantitative analyses and comparative evaluations discussed in subsequent phases.

Appendix: Technical Details

This section provides technical details about how a kernel29 module work, exemplified by the dxGPT module and how the *_async.py core logic scripts work, exemplified by judge_semantic_async.py and - judge_severity_async.py

Summary of dxGPT

This module (src.dxGPT) provides functionality to generate differential diagnoses for clinical cases using various Large Language Models (LLMs). It leverages the lapin framework for asynchronous processing, configuration management, database interaction, and modularity, mirroring the structural patterns observed in other project components.

Key features:

- Asynchronous Processing: Uses lapin.utils.async_batch.process_all_batches for efficient handling of multiple LLM API calls.
- Database Integration: Fetches input cases from a database table (e.g., CasesBench) and stores generated diagnoses (raw response and parsed items) in relational tables (e.g.,LlmDifferentialDiagnosis, DifferentialDiagnosis2Rank).
- Configurable Models: Supports multiple LLM backends through lapin's configuration system (models/dxGPT_models.py).
- Modular Prompts: Uses lapin's PromptBuilder pattern (prompts/dxGPT_prompts.py) to define and manage different prompt strategies, selectable via configuration.
- Flexible Parsing: Includes parsers (parsers/dxGPT_parsers.py) to extract structured diagnosis information from LLM responses.
- Structured Execution: Follows a pattern where a simple runner script (run_dxGPT_async.py) invokes the main asynchronous logic (dxGPT_async.py).

Core Components

1. dxGPT_async.py: Contains the main asynchronous execution logic and helper functions.
    - set_settings(): Initializes DB session, AsyncModelHandler, and the selected PromptBuilder.
    - retrieve_and_make_prompts(): Fetches CasesBench records, builds prompt strings using the PromptBuilder, and prepares simple wrapper objects (DxGPTInputWrapper) containing the prompt text and necessary metadata for process_all_batches.
    - process_results(): Takes the list of dictionaries returned by process_all_batches, extracts the original item data and the LLM response (or error), parses the response based on the original prompt used, and stores results in the database via queries.dxGPT_queries.insert_differential_diagnoses.
    - main_async(): Orchestrates the overall process: loads config, calls set_settings, retrieve_and_make_prompts,

lapin.utils.async_batch.process_all_batches, and process_results, handling setup and cleanup (like DB session closing).
    - Includes a if __name__ == '__main__' block for direct testing/debugging.
2. run_dxGPT_async.py: Command-line interface runner.
    - Uses argparse to define arguments (--model, --prompt_alias, --num_samples, etc.).
    - Sets up logging.
    - Calls the main_async() function in dxGPT_async.py, passing the parsed arguments.
3. models/dxGPT_models.py: Defines lapin configuration classes inheriting from framework base classes (e.g., lapin.conf.openai_conf.OpenAIConfig).
4. prompts/dxGPT_prompts.py: Defines PromptBuilder classes (lapin.prompt_builder.base.PromptBuilder) registered with lapin (@register_prompt).
5. parsers/dxGPT_parsers.py: Contains functions to parse LLM outputs (e.g., universal_dif_diagnosis_parser, parse_top5_xml). JSON parsing logic is currently within process_results.
6. queries/dxGPT_queries.py: Provides database interaction functions:
    - get_model_id: Retrieves model ID from alias.
    - fetch_cases_from_db: Queries CasesBench table.
    - insert_differential_diagnoses: Inserts results into LlmDifferentialDiagnosis and DifferentialDiagnosis2Rank tables.
7. serialization/dxGPT_schemas.py: Defines Pydantic models for internal data structuring (used within dxGPT_async.py but not directly by the batch runner).

Workflow

1. Execution: The run_dxGPT_async.py script is executed with command-line arguments.
2. Argument Parsing: The runner parses arguments (model, prompt, filters, etc.).
3. Main Logic Invocation: The runner calls main_async() in dxGPT_async.py, passing the arguments.
4. Configuration Loading: main_async() loads the LapinConfig using lapin.core.config.load_config, applying arguments as overrides.
5. Setup: main_async() calls set_settings() to get a DB session, instantiate the AsyncModelHandler, and load the specified PromptBuilder.
6. Input Preparation: main_async() calls retrieve_and_make_prompts() which:
    a. Fetches cases from the DB using queries.dxGPT_queries.fetch_cases_from_db.
    b. For each case, builds the prompt string using the loaded PromptBuilder.
    c. Creates a list of DxGPTInputWrapper objects containing the final prompt text and necessary IDs.
7. Asynchronous Batch Processing: main_async() calls lapin.utils.async_batch.process_all_batches, passing the list of input wrappers, the prompt builder instance (potentially redundant if handler uses config, TBC by lapin usage), the handler instance, model alias, and batching parameters.

8. Result Processing: `main_async()` calls `process_results()` which iterates through the list of results returned by `process_all_batches`:
    a. Extracts the raw LLM response and original input data.
    b. Parses the response using the appropriate logic based on the `prompt_alias` stored in the input data.
    c. Calls `queries.dxGPT_queries.insert_differential_diagnoses` to save the raw response and parsed diagnoses to the database.
9. Cleanup: `main_async()` ensures the database session is closed in a `finally` block.
10. Completion: Logging indicates the number of successful operations and errors.

Database Interaction

- Input: Reads from `CasesBench` table (fields: `id`, `description`, `additional_info`, `hospital`, `case_identifier`). Also reads `Models` table.
- Output:
    - Creates a record in `LlmDifferentialDiagnosis` (fields: `cases_bench_id`, `model_id`, `prompt_identifier`, `raw_response`).
    - Creates multiple linked records in `DifferentialDiagnosis2Rank` (fields: `differential_diagnosis_id`, `cases_bench_id`, `rank_position`, `predicted_diagnosis`, `reasoning`).

Configuration

Configuration is managed by `lapin`. Key aspects:

- Model Config: Defined in `models/dxGPT_models.py`, selected via `--model` argument.
- Prompt Builder: Defined in `prompts/dxGPT_prompts.py`, selected via `--prompt_alias` argument, which sets `params.prompt_alias` in config.
- Parameters: Runtime parameters (`num_samples`, filters, batching settings) passed via CLI arguments and merged into `config.params`.
- API keys/secrets handled by `lapin` config loading.

Summary of `judge_semantic_async.py`

This script evaluates the semantic relationship between LLM-generated differential diagnoses and the corresponding golden standard diagnosis for clinical cases stored in a database. It uses another LLM (the "semantic judge") to assess each ranked diagnosis produced by a target LLM against the known correct diagnosis.

Purpose

The primary goal is to asynchronously process numerous LLM-generated differential diagnoses, judge the semantic relationship (e.g., Exact Synonym, Broad Synonym, Related Disease Group, Not Related) of each ranked diagnosis relative to the golden standard diagnosis using a designated semantic judge model, and store these relationship judgments back into the database. This allows for analysis of the clinical relevance and accuracy of the LLM's diagnostic suggestions.

General Technical Overview

Here's a high-level overview of what this script does: 1. Connects to a database containing clinical cases, their correct ("golden standard") diagnoses, and differential diagnoses previously generated by different AI models. 2. Selects differential diagnoses generated by a specific AI model (e.g., 'llama2_7b') for a particular dataset (e.g., 'ramedis'). 3. For each case, it formats a text prompt containing both the golden standard diagnosis and the ranked list of diagnoses generated by the target AI model. 4. Sends these prompts asynchronously (many at once) to another AI model (the "semantic judge", e.g., 'llama3-70b'). 5. Receives the semantic relationship judgments (e.g., "Exact synonym", "Related Disease Group", "Not related") back from the semantic judge AI, indicating how closely each AI-generated diagnosis matches the golden standard. 6. Parses these text-based judgments into a structured format. 7. Stores the structured semantic relationship results back into the database.

The script can run using command-line arguments for automation (Endpoint Mode) or using fixed settings coded directly into the script for debugging (Debug Mode).

Detailed Technical Overview

Execution Modes

The script supports two execution modes, determined by the presence of the `--semantic_judge` command-line argument:

1. Endpoint Mode:
   - Triggered when `--semantic_judge` (and potentially other arguments like `--differential_diagnosis_model`, `--batch_size`, etc.) is provided.
   - Uses command-line arguments to configure settings (models, batching parameters, etc.).
   - Follows a structured, refactored execution path using the `main` function and helper functions (`set_settings`, `retrieve_and_make_prompts`, `process_results`).
   - This is the intended mode for automated or production-like runs.
2. Debug Mode:
   - Triggered when `--semantic_judge` is not provided.
   - Uses hardcoded settings defined directly within the `if __name__ == "__main__":` block.
   - Executes the core logic sequentially within the `else` block, without calling the refactored `main` or helper functions. The code in this block remains untouched from its original state for debugging and comparison purposes.

Core Logic Flow (Endpoint Mode)

When run in Endpoint mode, the script follows these steps orchestrated by the `main` function:

1. Argument Parsing: Reads command-line arguments to set parameters.

2. Settings Initialization (`set_settings`):
   - Establishes a database session (`get_session`).
   - Initializes the asynchronous API handler (`AsyncModelHandler`).
   - Loads the specified prompt template for the semantic judge (e.g., `Semantic_prompt()`).
3. Data Retrieval and Transformation (`retrieve_and_make_prompts`):
   - Fetches relevant ranked diagnosis records (`DifferentialDiagnosis2Rank` objects) from the database based on the target `differential_diagnosis_model` and `test_name` (hospital).
   - Fetches the golden standard diagnosis for each relevant case (`get_cases`, `get_case_to_golden_diagnosis_mapping`).
   - Groups the flat rank records into a list of nested dictionaries, where each dictionary represents one full differential diagnosis set for a specific case/model (`create_nested_diagnosis_dict`).
   - Formats each nested dictionary into a single plain text string containing both the golden diagnosis and the ranked LLM diagnoses, mapped to a unique composite key (`f"{case_id}_{model_id}_{diff_diag_id}"`) in a flat dictionary (`dif_diagnosis_dict2plain_text_dict_with_real_diagnosis`).
   - Wraps each key-value pair from the flat dictionary into a simple object (`DiagnosisTextWrapper`) having `.id` (the composite key) and `.text` (golden + ranked diagnoses string) attributes (`convert_dict_to_objects`). This final list of wrapper objects is required by the batch processing library (`process_all_batches`).
   - Also creates an intermediate dictionary (`ranked_differential_diagnosis_nested_dict`) mapping the composite keys back to the original rank details (`nested_dict2rank_dict`), used later for result parsing.
4. Asynchronous Batch Processing (`process_all_batches`):
   - Takes the list of `DiagnosisTextWrapper` objects.
   - Uses the `AsyncModelHandler` and the loaded prompt template (`semantic_prompt_builder`).
   - Sends requests to the `semantic_judge` model API in batches, respecting rate limits (`rpm_limit`, `min_batch_interval`). The prompt likely asks the judge to compare the golden diagnosis to each ranked diagnosis in the text.
   - Collects the raw text responses from the semantic judge model.
5. Result Processing (`process_results`):
   - Iterates through the raw results from the batch processing step.
   - Uses `parse_judged_semantic` (from `bench29.libs.judges.semantic.parsers.parser_libs`) to attempt parsing the structured semantic relationship judgments (e.g., for each ranked diagnosis: semantic category, reasoning) from the model's text response. It uses the `ranked_differential_diagnosis_nested_dict` to link results back to the original data if needed by the parser.
   - Separates successfully parsed results from failures.
6. Database Storage (`add_semantic_results_to_db`):

- Takes the list of successfully parsed semantic relationship judgments.
- Inserts these judgments into the appropriate database table (e.g., `differential_diagnosis_to_semantic_relationship` or a dedicated results table).

7. Statistics and Exit: Calculates and prints timing and throughput statistics. Waits for user input before exiting (in both modes).

Key Functions (Endpoint Mode)

This section details the core functions used in Endpoint Mode.

```
set_settings(prompt_name)
"""Initializes database session, asynchronous handler, and prompt builder.

Args:
   prompt_name (str): The name of the prompt class/function to use (e.g., 'Semantic_prompt').

Returns:
   tuple: A tuple containing:
      - session: SQLAlchemy database session object.
      - handler: An instance of AsyncModelHandler for API calls.
      - semantic_prompt_builder: The instantiated prompt builder object.
"""

retrieve_and_make_prompts(differential_diagnosis_model, test_name, session,
verbose, max_diagnoses)
"""Retrieves differential diagnoses and golden diagnoses, processes them, and prepares
objects for batching.

This function orchestrates the data preparation pipeline for semantic comparison:
1. Fetches individual ranked diagnosis entries (`DifferentialDiagnosis2Rank` objects)
  from the database based on the specified model and test name (hospital).
2. Fetches the corresponding golden standard diagnosis for each case involved.
3. Groups the ranked diagnoses into a list of nested dictionaries using
  `create_nested_diagnosis_dict`. Each nested dictionary represents a complete
  differential diagnosis set for a specific case/model combination.
4. Transforms this list into a flat dictionary using
`dif_diagnosis_dict2plain_text_dict_with_real_diagnosis`.
  The keys are composite strings (`f"{case_id}_{model_id}_{diff_diag_id}"`), and the values are
  multi-line strings containing BOTH the golden diagnosis AND the formatted list of ranked
  LLM diagnoses, separated by a specific string.
5. Converts this flat dictionary into a list of `DiagnosisTextWrapper` objects using
  `convert_dict_to_objects`. Each object has `.id` (composite key) and `.text` (golden + ranked
diagnoses).
  This format is required by `process_all_batches`.
6. Creates an intermediate dictionary mapping composite keys back to the original rank
details
```

using `nested_dict2rank_dict`, needed for result parsing.

Args:
differential_diagnosis_model (str): The name of the model whose diagnoses are being judged.
test_name (str): The identifier for the test run (e.g., hospital name).
session: SQLAlchemy database session object.
verbose (bool): If True, prints detailed progress messages.
max_diagnoses (int | None): Maximum number of diagnosis sets to process. If None, process all.

Returns:
tuple: A tuple containing:
- ranked_differential_diagnosis_objects (list): List of `DiagnosisTextWrapper` objects ready for batching.
- ranked_differential_diagnosis_nested_dict (dict): Intermediate dictionary mapping composite keys to rank details.
"""

```
process_results(results, ranked_differential_diagnosis_nested_dict, session, verbose, semantic_categories=None)
```
"""Processes the raw results from the batch API calls, parses semantic relationship judgments.

Iterates through the results obtained from `process_all_batches`. For each result,
it attempts to parse the semantic relationship judgment using `parse_judged_semantic`.
It separates successfully parsed results from failures.

Args:
results (list): The list of result dictionaries returned by `process_all_batches`.
ranked_differential_diagnosis_nested_dict (dict): The intermediate dictionary mapping composite keys
to original rank details, used by the parser to link results back to original data.
session: SQLAlchemy database session object.
verbose (bool): If True, prints warnings for results that couldn't be parsed.
semantic_categories (set | None): A set of valid semantic category strings.
Defaults if None.

Returns:
tuple: A tuple containing:
- semantic_judge_results (list): A list of successfully parsed semantic results.
- semantic_judge_fails (list): A list of results that failed parsing.
"""

```
main(verbose, semantic_judge, differential_diagnosis_model, batch_size,
max_diagnoses, rpm_limit, min_batch_interval, test_name, prompt_name)
"""Main execution logic for running the semantic judge in Endpoint mode.

Orchestrates the process: setup, data retrieval/transformation,
batch API calls for semantic judgment, result parsing, DB storage, and stats.

Args:
    verbose (bool): Enable detailed logging.
    semantic_judge (str): Model name for the semantic judge API.
    differential_diagnosis_model (str): Model name used for the diagnoses being judged.
    batch_size (int): Number of diagnoses per API call batch.
    max_diagnoses (int | None): Maximum number of diagnoses to process.
    rpm_limit (int): Requests per minute limit for the API.
    min_batch_interval (float): Minimum time between batches in seconds.
    test_name (str): Name for the test run.
    prompt_name (str): Name of the prompt configuration to use.
"""
```

Key Dependencies / Libraries

- Standard: `asyncio`, `time`, `argparse`, `sys`, `os`
- Database: `sqlalchemy` (implicitly via `db.utils.db_utils` and query functions)
- Asynchronous API Handling: `lapin` library (`AsyncModelHandler`, `process_all_batches`)
- Internal Bench29 Libs:
  - `bench29.libs.queries.common_queries`: `get_cases`, `get_case_to_golden_diagnosis_mapping`, `get_model_names_from_differential_diagnosis`, `get_ranks_for_hospital_and_model_id`, `get_model_id_from_name`, `create_nested_diagnosis_dict`.
  - `bench29.libs.queries.semantic_queries`: `add_semantic_results_to_db`.
  - `bench29.libs.utils.text_conversion`: `nested_dict2rank_dict`, `dif_diagnosis_dict2plain_text_dict_with_real_diagnosis`, `convert_dict_to_objects`.
  - `bench29.libs.judges.prompts.semantic_judge_prompts`: Contains prompt classes/functions (e.g., `Semantic_prompt`).
  - `bench29.libs.judges.semantic.parsers.parser_libs`: `parse_judged_semantic`.
- Database Utilities: `db.utils.db_utils`: `get_session`
- Path Setup: The script includes manual `sys.path` manipulation using `os` and `sys`.

Summary of `judge_severity_async.py`

This script evaluates the severity of differential diagnoses generated by Large Language Models (LLMs) for clinical cases stored in a database. It uses another LLM (the "severity judge") to assess each diagnosis within a ranked list produced by a target LLM.

Purpose

The primary goal is to asynchronously process numerous LLM-generated differential diagnoses, judge the severity of each individual diagnosis within those ranked lists using a designated severity judge model, and store these severity judgments back into the database. This allows for analysis of whether LLMs generate diagnoses of appropriate severity compared to the original case or potentially a gold standard.

General Technical Overview

Here's a high-level overview of what this script does: 1. Connects to a database containing clinical cases and diagnoses previously generated by different AI models. 2. Selects diagnoses generated by a specific AI model (e.g., 'llama2_7b') for a particular dataset (e.g., 'ramedis'). 3. Formats the ranked list of diagnoses for each case into a text prompt suitable for another AI model (the "severity judge", e.g., 'llama3-70b'). 4. Sends these prompts asynchronously (many at once) to the severity judge AI. 5. Receives the severity judgments (e.g., "mild", "severe", "critical") back from the AI. 6. Parses these text-based judgments into a structured format. 7. Stores the structured severity results back into the database.

The script can run using command-line arguments for automation (Endpoint Mode) or using fixed settings coded directly into the script for debugging (Debug Mode).

Detailed Technical Overview

Execution Modes

The script supports two execution modes, determined by the presence of the `--severity_judge` command-line argument:

1. Endpoint Mode:
   - Triggered when `--severity_judge` (and potentially other arguments like `--differential_diagnosis_model`, `--batch_size`, etc.) is provided.
   - Uses command-line arguments to configure settings (models, batching parameters, etc.).
   - Follows a structured, refactored execution path using the `main` function and helper functions (`set_settings`, `retrieve_and_make_prompts`, `process_results`).
   - This is the intended mode for automated or production-like runs.
2. Debug Mode:
   - Triggered when `--severity_judge` is not provided.

- Uses hardcoded settings defined directly within the `if __name__ == "__main__":` block.
- Executes the core logic sequentially within the `else` block, without calling the refactored `main` or helper functions. The code in this block remains untouched from its original state for debugging and comparison purposes.

Core Logic Flow (Endpoint Mode)

When run in Endpoint mode, the script follows these steps orchestrated by the `main` function:

1. Argument Parsing: Reads command-line arguments to set parameters.
2. Settings Initialization (`set_settings`):
   - Establishes a database session (`get_session`).
   - Initializes the asynchronous API handler (`AsyncModelHandler`).
   - Loads the specified prompt template for the severity judge (`eval(prompt_name)()`).
3. Data Retrieval and Transformation (`retrieve_and_make_prompts`):
   - Fetches relevant ranked diagnosis records (`DifferentialDiagnosis2Rank` objects) from the database based on the target `differential_diagnosis_model` and `test_name` (hospital).
   - Groups these flat records into a list of nested dictionaries, where each dictionary represents one full differential diagnosis set for a specific case/model (`create_nested_diagnosis_dict`).
   - Formats each nested dictionary into a single plain text string containing the ranked diagnoses, mapped to a unique composite key (`f"{case_id}_{model_id}_{diff_diag_id}"`) in a flat dictionary (`dif_diagnosis_dict2plain_text_dict`).
   - Wraps each key-value pair from the flat dictionary into a simple object (`DiagnosisTextWrapper`) having `.id` (the composite key) and `.text` (the formatted diagnosis list) attributes (`convert_dict_to_objects`). This final list of wrapper objects is required by the batch processing library (`process_all_batches`).
   - Also creates an intermediate dictionary (`ranked_differential_diagnosis_nested_dict`) mapping the composite keys back to the original rank details (`nested_dict2rank_dict`), used later for result parsing.
4. Asynchronous Batch Processing (`process_all_batches`):
   - Takes the list of `DiagnosisTextWrapper` objects.
   - Uses the `AsyncModelHandler` and the loaded prompt template.
   - Sends requests to the `severity_judge` model API in batches, respecting rate limits (`rpm_limit`, `min_batch_interval`).
   - Collects the raw text responses from the severity judge model.
5. Result Processing (`process_results`):
   - Iterates through the raw results from the batch processing step.

- Uses `parse_judged_severity` (from `bench29.libs.judges.severity.parsers.parser_libs`) to attempt parsing the structured severity judgment (disease, severity level, rank, reasoning) from the model's text response. It uses the `ranked_differential_diagnosis_nested_dict` to link results back to the original data if needed by the parser.
- Separates successfully parsed results from failures.

6. Database Storage (`add_severity_results_to_db`):
   - Takes the list of successfully parsed severity judgments.
   - Inserts these judgments into the appropriate database table (e.g., `differential_diagnosis_to_severity` or a dedicated results table).
7. Statistics and Exit: Calculates and prints timing and throughput statistics. Waits for user input before exiting (in both modes).

Key Functions (Endpoint Mode)

This section details the core functions used in Endpoint Mode.

```
set_settings(prompt_name)
"""Initializes database session, asynchronous handler, and prompt builder.

Args:
   prompt_name (str): The name of the prompt function to use (e.g., 'prompt_1').

Returns:
   tuple: A tuple containing:
      - session: SQLAlchemy database session object.
      - handler: An instance of AsyncModelHandler for API calls.
      - severity_prompt_builder: The instantiated prompt builder object.
"""

retrieve_and_make_prompts(differential_diagnosis_model, test_name, verbose,
max_diagnoses)
"""Retrieves differential diagnoses, processes them, and prepares objects for batching.

This function orchestrates the data preparation pipeline:
1. Fetches individual ranked diagnosis entries (`DifferentialDiagnosis2Rank` objects)
  from the database based on the specified model and test name (hospital).
  Each object contains fields like `cases_bench_id`, `differential_diagnosis_id`,
  `rank_position`, and `predicted_diagnosis`.
2. Groups these flat rank objects into a list of nested dictionaries using
  `create_nested_diagnosis_dict`. Each nested dictionary represents a complete
  differential diagnosis set for a specific case/model combination and contains
  a list of its ranked diagnoses.
  Example structure: `[{'model_id': M, 'cases_bench_id': C, 'differential_diagnosis_id': D,
            'ranks': [{'rank_id': R1, 'rank': 1, 'predicted_diagnosis': 'DiagA'}, ...]}]
3. Transforms this list of nested dictionaries into a flat dictionary using
```

```
    `dif_diagnosis_dict2plain_text_dict`. The keys of this flat dictionary are composite
    strings (`f"{C}_{M}_{D}"`) uniquely identifying each differential diagnosis set,
    and the values are multi-line strings containing the formatted list of diagnoses
    (e.g., "- DiagA\n- DiagB").
4. Converts this flat dictionary into a list of simple objects (`DiagnosisTextWrapper`)
    using `convert_dict_to_objects`. Each object has an `.id` attribute (the composite key
    string from step 3) and a `.text` attribute (the formatted diagnosis string from step 3).
    This object format (`list[object]` where `object.id` and `object.text` exist) is required
    by the `lapin.utils.async_batch.process_all_batches` function.

Args:
    differential_diagnosis_model (str): The name of the model whose diagnoses are being
judged.
    test_name (str): The identifier for the test run (e.g., hospital name).
    verbose (bool): If True, prints detailed progress messages.
    max_diagnoses (int | None): Maximum number of diagnosis *sets* (nested dicts) to process.
        Note: This limits based on the output of `create_nested_diagnosis_dict` before
        conversion to text/objects. If None, process all.

Returns:
    tuple: A tuple containing:
        - ranked_differential_diagnosis_objects (list): The final list of `DiagnosisTextWrapper`
objects
          ready for `process_all_batches`.
        - ranked_differential_diagnosis_nested_dict (dict): The intermediate dictionary mapping
composite
          keys to the list of rank dictionaries (`{'rank_id': R, 'rank': Pos, 'predicted_diagnosis':
Diag}`),
          created by `nested_dict2rank_dict`. This is used later for result parsing.
"""

process_results(results, ranked_differential_diagnosis_nested_dict, session,
verbose, severity_levels=None)
"""Processes the raw results from the batch API calls, parses severity judgments.

Iterates through the results obtained from `process_all_batches`. For each result,
it attempts to parse the severity judgment using `parse_judged_severity`.
It separates successfully parsed results from failures.

Args:
    results (list): The list of result dictionaries returned by `process_all_batches`.
    ranked_differential_diagnosis_nested_dict (dict): The nested dictionary containing original
        diagnosis details, used by the parser to link results back to original data.
    session: SQLAlchemy database session object.
    verbose (bool): If True, prints warnings for results that couldn't be parsed.
    severity_levels (set | None): A set of valid severity level strings.
```

```
    Defaults to {"rare", "critical", "severe", "moderate", "mild"} if None.

Returns:
    tuple: A tuple containing:
        - severity_judge_results (list): A list of successfully parsed severity results
          (likely objects or dictionaries ready for database insertion).
        - severity_judge_fails (list): A list of results that failed parsing.
"""

main(verbose, severity_judge, differential_diagnosis_model, batch_size,
max_diagnoses, rpm_limit, min_batch_interval, test_name, prompt_name)
"""Main execution logic for running the severity judge in Endpoint mode.

This function orchestrates the entire process when the script is run
with command-line arguments (Endpoint mode). It sets up resources,
retrieves and transforms data, runs batch processing via API calls,
parses the results, stores them in the database, and prints statistics.

Args:
    verbose (bool): Enable detailed logging.
    severity_judge (str): Model name for the severity judge API.
    differential_diagnosis_model (str): Model name used for the diagnoses being judged.
    batch_size (int): Number of diagnoses per API call batch.
    max_diagnoses (int | None): Maximum number of diagnoses to process.
    rpm_limit (int): Requests per minute limit for the API.
    min_batch_interval (float): Minimum time between batches in seconds.
    test_name (str): Name for the test run.
    prompt_name (str): Name of the prompt configuration to use.
"""
```

Key Dependencies / Libraries

- Standard: `asyncio`, `time`, `argparse`, `sys`, `os`, `json`
- Database: `sqlalchemy` (implicitly via `db.utils.db_utils` and query functions)
- Asynchronous API Handling: `lapin` library (`AsyncModelHandler`, `process_all_batches`)
- Internal Bench29 Libs:
  - `bench29.libs.queries.severity_queries`: `get_severity_id`, `add_severity_results_to_db`
  - `bench29.libs.queries.common_queries`: `get_model_names_from_differential_diagnosis`, `get_ranks_for_hospital_and_model_id`, `get_model_id_from_name`, `create_nested_diagnosis_dict`
  - `bench29.libs.judges.prompts.severity_judge_prompts`: Contains prompt templates (e.g., `prompt_1`).

    - `bench29.libs.utils.text_conversion`: `dif_diagnosis_dict2plain_text_dict`, `convert_dict_to_objects`, `nested_dict2rank_dict`.
    - `bench29.libs.judges.severity.parsers.parser_libs`: `parse_judged_severity`.
- Database Utilities: `db.utils.db_utils`: `get_session`
- Initialization: `__init__` (Likely handles path setup or other initializations).

CHAPTER 6: Deep Dive into `lapin` Internals: Orchestration Mechanics

Introduction

Previous chapters established `lapin` as the cornerstone of Kernel29's LLM orchestration strategy, handling both synchronous and asynchronous interactions. This chapter moves beyond the conceptual overview (Chapter 4) to provide a low-level, technical examination of `lapin`'s core components, excluding the `PromptBuilder`. We will dissect the specific classes, methods, and design patterns within `lapin.conf`, `lapin.callers`, `lapin.handlers`, `lapin.utils.async_batch`, and `lapin.trackers` that enable robust configuration, unified API interaction, and efficient execution. This detailed analysis is crucial for developers extending Kernel29 or debugging interactions within the framework.

Configuration Loading and the Component Registry (`lapin.conf`)

The foundation of `lapin`'s flexibility lies in its configuration management, centered around `lapin.conf` and a dynamic registry.

Configuration Classes (`LapinConfig` and Subclasses):

`lapin.conf` defines base configuration classes (e.g., `base_conf.BaseLapinConfig`). Specific provider implementations (e.g., `groq_conf.GroqConfig`, `anthropic_conf.AnthropicConfig`) inherit from these bases. These subclasses define fields essential for API interaction:

- Credentials: API keys, secrets (often loaded from environment variables, potentially using Pydantic).
- Endpoints: Base URLs, specific API versions, or deployment IDs.
- Model Identification: The specific model name required by the provider's API (e.g., `gemma-7b-it`, `claude-3-opus-20240229`).
- API Parameters: Default values and types for parameters like `temperature`, `max_tokens`.
- Caller Specification: Methods like `caller_class()` and `async_caller_class()` that return the appropriate synchronous or asynchronous caller class for this configuration.
- Tracker Specification: A method like `tracker_class()` that returns the appropriate usage tracker class.

The Registry (`@register_config`):

The `@register_config("alias_name")` decorator, used in `lapin/conf/__init__.py` or within specific config files, populates a global dictionary (e.g., `CONFIG_REGISTRY` in `lapin.conf.base_conf`). It maps the string `alias_name` to the configuration class itself (`alias_name`: `ConfigClass`).

1. **Instantiation**: During the `set_settings` phase of a Kernel29 module (Chapter 5) or when a handler is initialized, the alias string (e.g., `--model alias_name` from CLI

args) is used to look up the corresponding `ConfigClass` in the registry. An instance is created (e.g., `config_obj = config_cls()`), holding validated connection and parameter details.

API-Specific Communication (`lapin.callers`)

Before the handler can orchestrate a call, there needs to be code that *actually* communicates with each specific LLM provider's API. This is the role of the **Callers**.

Modules within `lapin.callers` (e.g., `sync_groq_caller.py`, `async_groq_caller.py`, `anthropic_caller.py`) contain the low-level, provider-specific API interaction logic. Think of them as specialized workers, each expert in talking to one particular service.

- **Internal Components**: Callers are **internal components** used by the handlers and are typically **not directly called** by application code.
- **Sync/Async Implementations**: Separate classes exist for synchronous and asynchronous operations:
  - **Sync Callers**: Implement a method like `call_llm(self, prompt: str)`. They use standard blocking HTTP libraries (like `requests`) or synchronous SDK methods.
  - Async Callers: Implement a method like `async call_llm_async(self, prompt: str)`. They use non-blocking, asynchronous HTTP libraries (like `httpx`, `aiohttp`) or async SDK methods.

Core Responsibilities:

Both caller types are responsible for:

- Accepting API parameters during instantiation (`__init__(self, params)`).
- Constructing the exact request payload (e.g., JSON body) and headers (including authentication) required by the specific provider's API.
- Making the actual HTTP request to the correct endpoint.
- Handling the provider-specific response format (e.g., parsing JSON responses).
- Basic Parsing: Extracting the core generated text content from the complex raw API response object.
- Returning both the full, raw response object (which might contain metadata like token usage) and the extracted, parsed text back to the handler.
- Managing provider-specific errors and potentially raising exceptions.

By encapsulating the direct API interaction logic here, the rest of `lapin` (especially the handlers) doesn't need to be concerned with the unique details of each provider's protocol.

Unified Orchestration Facade (`lapin.handlers`)

With specialized Callers ready to handle specific API communications, the **Handlers** (`AsyncModelHandler`, `ModelHandler` located in `lapin.handlers`) act as the central **orchestration engine** and **facade**. They provide a **single, unified interface** for the application code, hiding the complexity of the underlying components (Configs, Callers, Trackers).

Design Rationale and Advantages: The Handler's role as a facade provides significant, concrete advantages for the framework's structure and usability:

- **Decoupling**: Application code (like `dxGPT_async.py`) interacts *only* with the Handler's simple `get_response(alias, prompt)` method. It remains completely unaware of which specific Caller is used, how authentication works for different providers, or how rate limits are tracked. This means adding support for a new LLM provider (by adding a new Config and Caller) requires **no changes** to the application code that consumes the Handler.
- **Centralized Workflow Control**: The Handler enforces a consistent sequence of operations for *every* LLM call: look up config, get caller, get tracker, check rate limits, execute call via caller, record usage via tracker. This standardizes the interaction lifecycle and prevents crucial steps like rate limiting or usage tracking from being accidentally omitted in different parts of the application.
- **Simplified Application Logic**: The application avoids complex conditional logic (`if/elif/else`) to handle different providers. It delegates the provider-specific selection and execution details entirely to the Handler based on the `alias`, leading to cleaner, more focused application code.
- **Maintainability**: All the core orchestration logic (the sequence of coordinating Configs, Callers, and Trackers) is centralized within the Handlers. This makes the primary LLM interaction workflow easier to understand, debug, and modify if

needed, without impacting unrelated application logic or specific Caller implementations.

Implementation Details: Handlers achieve this orchestration as follows:

1. **Gateway Methods**: They expose the primary methods for application code:
   - `AsyncModelHandler.get_response(self, prompt: str, alias: str, only_text: bool = True)` (asynchronous)
   - `ModelHandler.get_response(self, prompt: str, alias: str, only_text: bool = True)` (synchronous)
2. **Internal Orchestration Flow** (within `get_response`):
   - **Lookup & Validation**: Finds the `ConfigClass` for the given `alias` in the `CONFIG_REGISTRY`.
   - **Component Instantiation**: Creates instances of:
     - The specific `LapinConfig` subclass (`config_obj = config_cls()`).
     - The appropriate Caller (sync or async) specified by the config (`caller = config_obj.caller_class()(params)` or `caller = config_obj.async_caller_class()(params)`).
     - The correct usage Tracker specified by the config (`model_tracker = config_obj.tracker_class().get_model(...)`).
   - **Pre-call Check**: Consults the `model_tracker` via `should_pause()` to ensure rate limits are respected.
   - **Execution Delegation**: Invokes the core execution method on the instantiated Caller (`caller.call_llm(prompt)` or `await caller.call_llm_async(prompt)`), passing the prompt and receiving back the `response` and `parsed_response`.
   - **Post-call Tracking**: Uses the `response` object to record usage details via the `model_tracker` (`record_request_by_provider` or `record_request`).
   - **Result Return**: Returns the results (`parsed_response` or `(response, parsed_response)`) back to the calling application code.

In essence, the application tells the Handler *what* it wants (which model alias, what prompt), and the Handler figures out *how* to get it done by coordinating the Configs, Callers, and Trackers.

Asynchronous Batch Processing (`lapin.utils.async_batch`)

For efficient handling of multiple asynchronous requests, `lapin` provides `lapin.utils.async_batch.process_all_batches`.

1. **Input**: Takes a list of items, a `prompt_template` object (from `lapin.prompt_builder`), an initialized `AsyncModelHandler` instance, `model` alias, batching parameters (`batch_size`, `rpm_limit`, `min_batch_interval`), and attributes identifying text/ID within items.
2. **Task Creation (`make_prompt_tasks`)**: For each item in a batch, it uses an internal helper (`async_prompt_processing`) to create an `asyncio.Task`. This helper function

takes an item, generates the prompt using `prompt_template.to_prompt(text)`, calls the handler's `get_response` method, and returns a structured dictionary containing the result, tokens used, and timing.

3. **Concurrent Execution**: Uses `asyncio.gather(*tasks)` to execute all tasks for a batch concurrently.
4. **Rate Limiting (`should_wait`)**: Before starting a new batch, it calculates if a pause is needed based on `min_batch_interval` and the end time of the previous batch, calling `await asyncio.sleep(wait_time)` if necessary.
5. **Result Aggregation**: Collects the dictionaries returned by `asyncio.gather` for each batch into a final list.
6. **Output**: Returns the aggregated list of result dictionaries.

Note: There is no equivalent general-purpose synchronous batch processing utility within `lapin.utils`. Synchronous operations are typically handled individually via `ModelHandler` or require custom looping logic outside `lapin`.

Usage Tracking (`lapin.trackers`)

The `lapin.trackers` module facilitates monitoring, primarily for rate limiting and cost analysis.

1. **`BaseTracker` (`base_tracker.py`)**: An abstract base class defining the core interface and common logic for tracking requests and tokens within time windows (last minute, last day). It includes methods like `record_request`, `check_rate_limits`, `should_pause`, and `prompt2price`.
2. **Provider-Specific Trackers** (e.g., `groq_tracker.GroqTracker`): Subclasses of `BaseTracker` that implement provider-specific details, such as:
   - Default rate limits (RPM, RPD, TPM, TPD).
   - Pricing information (`prompt_price`, `completion_price`, `price_scale`).
   - The `record_request_by_provider` method, which knows how to extract usage details (like tokens) directly from the specific provider's response object.
   - Class methods or structures to manage trackers per model (e.g., `get_model`).
3. **Integration**: Handlers (`AsyncModelHandler`, `ModelHandler`) retrieve the appropriate tracker instance based on the configuration (`config_obj.tracker_class().get_model(...)`) and call its recording methods (`record_request_by_provider` or `record_request`) after each API call.

Conclusion

`lapin` orchestrates LLM interactions through structured configuration (`lapin.conf`), specialized API communication modules (`lapin.callers`), and a unifying facade (`lapin.handlers`) that manages the workflow including rate limiting and usage tracking (`lapin.trackers`). The Handler/Caller pattern is key to `lapin`'s extensibility and maintainability, decoupling application logic from provider specifics. Asynchronous operations can be efficiently scaled using the utility in `lapin.utils.async_batch`. Key

methods like `ModelHandler.get_response`, `AsyncModelHandler.get_response`, and the internal caller methods (`call_llm`, `call_llm_async`) drive the interaction flow. This technical design provides the flexibility and robustness necessary for the scalable and reproducible experimentation workflows central to the Kernel29 framework.

CHAPTER 7: Dynamic Prompt Engineering with `lapin.prompt_builder`

Introduction

Effective Large Language Model (LLM) interaction hinges significantly on prompt engineering. As highlighted in Chapter 1 (Table 4), hardcoding prompts or scattering prompt logic across application code leads to inflexibility, poor maintainability, and difficulty in systematic experimentation. `lapin` addresses this challenge through its `prompt_builder` component, providing a structured and dynamic approach to defining, managing, and utilizing prompt strategies.

This chapter delves into the technical implementation of the `lapin.prompt_builder`, focusing on the `PromptBuilder` base class found in `lapin/prompt_builder/base.py`. We will examine how it enables the separation of prompt construction logic from core application code, facilitates the assembly of prompts from various sources, and integrates with the `lapin` registry system for dynamic selection of prompt strategies at runtime.

Core Concept: The `PromptBuilder` Base Class

The foundation of the system is the abstract base class `lapin.prompt_builder.base.PromptBuilder`. Subclasses inheriting from `PromptBuilder` encapsulate the logic for constructing a specific type of prompt.

Key Characteristics and Methods:

Section-Based Composition:

`PromptBuilder` allows prompts to be built from distinct named "sections". These sections represent reusable parts of a prompt (e.g., instructions, context definitions, examples, output format specifications).

Flexible Section Loading:

The base class provides methods to load content for these sections from various sources during the builder's initialization:

- `load_section_from_db(...)`: Fetches section content stored in a database (likely via `bench29.libs.prompt_libs.get_prompt`).
- `load_section_from_file(...)`: Loads section content from a file path.
- `load_section_from_table(...)`: Loads and transforms data from a database table.
- `load_section_from_text(...)`: Uses a provided string directly (e.g., for default instructions).
- `set_section(...)`: Directly assigns content to a section. Loaded section content is stored in the `self.sections` dictionary.

Meta Template (`set_meta_template`):

Each builder defines a main template string (`self.meta_template`). This string uses

standard Python f-string/`.format()` style placeholders (`{placeholder_name}`) to indicate where:

- Static section content (`self.sections`) should be inserted.
- Dynamic, runtime arguments (like the specific input text or case details) should be placed.

Initialization Logic (`initialize`):

This is an abstract method that must be implemented by each specific `PromptBuilder` subclass. The `initialize` method is responsible for:

- Calling `set_meta_template` to define the prompt's overall structure.
- Calling the various `load_section_*` methods to populate `self.sections` with the required static content for that specific prompt strategy.

Template Building (`build_template`):

This internal method orchestrates the first stage of prompt construction. It calls `build_partial_template`, which substitutes the static content from `self.sections` into the `self.meta_template`. The result (`self.prompt_template`) is a template string where static parts are filled in, but placeholders for dynamic runtime arguments remain.

Runtime Formatting (`to_prompt`):

This is the primary method called by application code to generate the final, fully formatted prompt string. It performs the following:

- Ensures `build_template` has been called (if not, it calls it).
- Takes the dynamic runtime data as input. This can be a single `text` argument (if only one dynamic placeholder like `{input_text}` remains in `self.prompt_template`) or a `kwargs` dictionary for multiple dynamic placeholders (`{case_narrative}`, `{patient_age}`, etc.).
- Uses Python's string `.format(**kwargs)` method on `self.prompt_template` to insert the dynamic runtime values into the remaining placeholders.
- Returns the final, complete prompt string ready to be sent to the LLM via a `lapin` Handler.

The Prompt Registry (`@register_prompt`)

Similar to how model configurations are managed (Chapter 7), `lapin` employs a registry for `PromptBuilder` subclasses.

Registration:
The `@register_prompt("alias_name")` decorator associates a unique string alias (e.g., `dxgpt_standard`, `judge_semantic_v1`) with a specific `PromptBuilder` subclass.

Dynamic Loading:
During the `set_settings` phase of a Kernel29 module (Chapter 5), the prompt alias provided via CLI arguments (e.g., `--prompt_alias dxgpt_standard`) is used to look up the corresponding `PromptBuilder` class in the registry.

Instantiation & Initialization:
An instance of the selected `PromptBuilder` subclass is created, and its `initialize()` method is called. This prepares the builder by loading its sections and setting its meta-template according to its specific strategy.

This registry mechanism allows users to easily switch between different, potentially complex prompt strategies at runtime simply by changing a command-line argument, facilitating systematic experimentation.

Integration with Kernel29 Workflow

The initialized `PromptBuilder` instance plays a crucial role within the standard Kernel29 module workflow (Chapter 5), typically within the `retrieve_and_make_prompts` function or similar data preparation steps:

1. **Data Fetching**: Input data (e.g., clinical cases from `CasesBench`) is retrieved from the database.
2. **Prompt Generation**: For each data item, the relevant dynamic information is extracted and passed to the `prompt_builder.to_prompt(kwargs=dynamic_data)` method. This generates the final prompt string.
3. **Wrapping**: The generated prompt string, along with necessary metadata, is wrapped in an input object (e.g., `DxGPTInputWrapper`) ready for batch processing.

In the context of asynchronous batching (`lapin.utils.async_batch`), the `process_all_batches` function utilizes the `prompt_template` object (which is the initialized `PromptBuilder` instance) within its `async_prompt_processing` helper to call `prompt_template.to_prompt(text)` for each item before passing the result to the `AsyncModelHandler`.

Advantages of the `PromptBuilder` Approach

This structured approach to prompt management offers significant technical advantages:

- **Separation of Concerns**: Prompt design and logic (defining templates, loading sections) are cleanly separated from the core application logic (data fetching, LLM interaction orchestration, result processing).
- **Reusability**: Common prompt sections (e.g., standard instructions, safety guidelines) can be defined once (in files or DB) and loaded by multiple `PromptBuilder` subclasses.
- **Consistency**: Ensures prompts for a given strategy are constructed identically every time, following the logic defined in the specific `PromptBuilder` subclass.
- **Experimentation Agility**: Switching between fundamentally different prompt strategies (e.g., zero-shot vs. few-shot, XML vs. JSON output format) is achieved by simply changing the `--prompt_alias` argument, without altering application code.
- **Maintainability**: Modifying or updating a specific prompt strategy only requires changes within the corresponding `PromptBuilder` subclass and its associated sections/meta-template.

- **Readability**: Complex prompt logic involving multiple parts and conditional formatting is encapsulated within the builder, making the main application code cleaner.

Conclusion

The `lapin.prompt_builder` component provides a robust and flexible system for managing prompt engineering within the Kernel29 framework. By combining a section-based loading mechanism, meta-templates, a clear initialization and formatting lifecycle (`initialize`, `build_template`, `to_prompt`), and integration with a dynamic registry (`@register_prompt`), it effectively decouples prompt logic from application code. This facilitates easier maintenance, promotes consistency, and enables rapid experimentation with different prompt strategies, complementing the configuration and execution capabilities of the `lapin` handlers and callers described in Chapter 7.