#Structuring the notebooks based on the collected and parsed metadata

```
ask_structured_schema = {
   "name": "ask_structured_schema",
   "description": (
      "**IMPORTANT**: Your *entire* response must be valid JSON matching this schema—**no** single-quotes, no Python `None`, no trailing commas, no code fences,\n"
      "From the competition metadata, dataset metadata, and a raw Jupyter notebook text, "
      "extract exactly these fields as JSON (no extra keys, no prose, no markdown):\n"
      "  - competition_problem_type: one of ['classification','regression']\n"
      "  - competition_problem_subtype: single, concise, lowercase-and-hyphenated phrase (e.g. “binary-classification”, “multiclass-classification”, “multi-label-classification”, “time-series-forecasting”, “continuous-regression”, “ordinal-regression”, etc. or any other that fits.)\n"
      "  - competition_problem_description: dense, short, factual description of the problem, what needs to be found, no repetitive words (omit dataset-meta here)\n"
      "  - dataset_metadata: plain-English dataset_metadata in plain English as a single coherent paragraph, removing any non-human symbols (no bullets or symbols)\n"
      "  - competition_dataset_type: one of ['Tabular','Time-series','Text','Image','Audio','Video','Geospatial','Graph','Multimodal']\n"
      "  - preprocessing_steps: array of strings, each describing one transformation (e.g. 'median-impute missing values')\n"
      "  - notebook_model_layers_code: literal code snippet that builds(e.g model.fit) each layer(e.g Dense, Conv, etc..) and compiles the model(e.g model.compile) \n"
      "  - used_technique: either 'DL' or 'ML'\n"
      "  - library: string naming the main library used (exactly one 'Tensorflow', 'Pytorch')\n"
      "  - target_column: array of all column names in the dataset that must be predicted \n"
      "Emit ONLY those keys."
   ),
   "parameters": {
      "type": "object",
      "properties": {
         "competition_problem_type": {
            "type": "string",
            "enum": ["classification", "regression"],
            "description": "Pick exactly one."
         },
         "competition_problem_subtype": {
            "type": "string",
            "enum": [
               "binary-classification",
               "multiclass-classification",
               "multi-label-classification",
               "time-series-forecasting",
```

```
            "continuous-regression",
            "quantile-regression",
            "multi-output-regression",
            "ordinal-regression",
            "missing-value-imputation"
          ],
          "description": "Pick exactly one."
        },
        "competition_problem_description": {
          "type": "string",
          "description": "Dense, factual description of what needs to be predicted."
        },
        "dataset_metadata": {
          "type": "string",
          "description": "Plain-English paragraph describing the dataset."
        },
        "competition_dataset_type": {
          "type": "string",
          "enum":
["Tabular","Time-series","Text","Image","Audio","Video","Geospatial","Graph","Multimodal"],
          "description": "Choose one primary modality."
        },
        "preprocessing_steps": {
          "type": "array",
          "items": {"type": "string"},
          "description": "List every transformation (scaling, normalization, one-hot encoding,
etc.) in plain English."
        },
        "notebook_model_layers_code": {
          "type": "string",
          "description": (
            "include the literal code lines of model compile, model fit, and that build each layer
(e.g. `Dense(128, activation='relu', …)`)\n"
            "The line(s) that create or instantiate the model (Sequential, Functional, subclass,
torch.nn.Module, etc.).\n"
            "All layer-construction calls (Dense, Conv2D, custom layers, etc.) or layer
definitions in a subclass.\n"
            "The call that compiles or configures training (e.g. `compile()`,
`configure_optimizers()`, etc.).\n"
            "The call that launches training (e.g. `fit()`, `trainer.fit()`, `train()`, etc.).\n"
            "Do not include unrelated code, helper wrappers, or omit any of these steps."
          )
        },
        "used_technique": {
```

```
            "type": "string",
            "enum": ["DL","ML"],
            "description": "Either 'DL' or 'ML'."
          },
          "library": {
            "type": "string",
            "description": "Name of the main library used."
          },
          "target_column": {
            "type": "array",
            "items": {"type": "string"},
            "description": "List of all column names in train.csv to predict."
          }
        },
        "required": [
          "competition_problem_type",
          "competition_problem_subtype",
          "competition_problem_description",
          "dataset_metadata",
          "competition_dataset_type",
          "preprocessing_steps",
          "notebook_model_layers_code",
          "used_technique",
          "library",
          "target_column"
        ]
      }
    }
```

#Building a Keras model using the metadata

```
tools = [
  {
    "name": "generate_keras_schema",
    "type": "function",
    "description": (
      "***Generate and save a runnable deep learning model using Keras in Python code wrapped in <Code>…</Code> in a single `notebook_code` JSON field:\\n"
      "The generated code must implement:\n"
      "1. **Reproducibility**: set seeds for Python, NumPy, scikit-learn, and TensorFlow (or PyTorch).\n"
      "2. **Imports**:\\n\"  \n"
      "   - `pandas`, `numpy`\\n\"  \n"
      "   - `sklearn.model_selection.train_test_split`\\n\"  \n"
      "   - `sklearn.impute.SimpleImputer`\\n\"  \n"
      "   - `sklearn.compose.ColumnTransformer`\\n\"  \n"
      "   - `sklearn.preprocessing.StandardScaler`,`OneHotEncoder`,`LabelEncoder`  ← **added here**\\n\"  \n"
      "   - `sklearn.pipeline.Pipeline`\\n\"  \n"
      "   - `tensorflow` (or `torch`)\\n\"  \n"
      "   - `tensorflow.keras.callbacks.EarlyStopping,ModelCheckpoint`\\n\"  \n"
      "   - `from tensorflow.keras.metrics import SparseTopKCategoricalAccuracy`\\n\"  \n"
      "   - `json`, `time`\\n\"  \n"
      "   When using OneHotEncoding, use sparse_output=False instead of sparse\n"
     "3. Data Loading, Split & Target Encoding:\n"
      "Read each file in files_list into train_dfs\n"
      "If any filename endswith 'test.csv', load it into df_test, else df_test=None\n"
      "Infer id_col & target_columns from submission_example header\n"
      "df = pd.concat(train_dfs, ignore_index=True)\n"
      "# Target encoding immediately after df is final:\n"
      "col = target_columns[0]\n"
      "if competition_problem_subtype in ['binary-classification']:\n"
      "    from sklearn.preprocessing import LabelEncoder\n"
      "    le=LabelEncoder().fit(df[col].astype(str))\n"
      "    y_enc=le.transform(df[col].astype(str)).astype(int)\n"
      "    classes_=le.classes_\n"
      "elif competition_problem_subtype in ['multiclass-classification','multiclass classification','ordinal-regression']:\n"
      "    from sklearn.preprocessing import LabelEncoder\n"
      "    le=LabelEncoder().fit(df[col].astype(str))\n"
      "    y_enc=le.transform(df[col].astype(str))\n"
      "    classes_=le.classes_\n"
```

```
"elif competition_problem_subtype in ['multi-label-classification']:\n"
"    from sklearn.preprocessing import MultiLabelBinarizer\n"
"    mlb=MultiLabelBinarizer()\n"
"    y_enc=mlb.fit_transform(df[target_columns])\n"
"    classes_=mlb.classes_\n"
"elif competition_problem_subtype in
['continuous-regression','quantile-regression','multi-output
regression','missing-value-imputation']:\n"
"    y_values = df[target_columns].astype(float).values\n"
"    y_enc = np.log1p(y_values) if np.all(y_values >= 0) else y_values\n"
"elif competition_problem_subtype in
['time-series-forecasting','multivariate-time-series-forecasting']:\n"
"    y_enc=df[target_columns].values\n"
"else:\n"
"    y_enc=df[target_columns].values\n"
"X=df.drop(columns=target_columns+[id_col],errors='ignore')\n"
"# now either use provided df_test or split off 20% for test:\n"
"if df_test is None:\n"
"    X_train,X_val,y_train,y_val=train_test_split(\n"
"        X,y_enc,\n"
"        test_size=0.2,\n"
"        stratify=y_enc if competition_problem_subtype in
['binary-classification','multiclass-classification','multiclass classification'] else None,\n"
"        random_state=42)\n"
"    train_ids=X_train[id_col]\n"
"    test_ids =X_val[id_col]\n"
"else:\n"
"    X_train=X\n"
"    y_train=y_enc\n"
"    train_ids=df[id_col]\n"
"    test_ids =df_test[id_col]\n"
"    X_val   =df_test.drop(columns=target_columns+[id_col],errors='ignore')\n"
"    y_val   = None  # explicitly set\n"
"\n"
"4. Feature Engineering:\n"
   "Automatically drop columns with all missing values\n"
   "Identify categorical columns and remove those with extremely high cardinality (eg
>50 unique)\n"
   "Optionally apply any additional simple transformations you deem useful\n"
"5. **Preprocessing Pipeline**:\n"
   "   - Auto-detect numeric vs. categorical via `df.select_dtypes`.\n"
   "   - Build a `ColumnTransformer` with median‑imputed & scaled numerics, and
most‑frequent‑imputed & OHE categoricals (cap at 50 cats).\n"
   "   - Fit on train → transform train/val/test.\n"
```

```
"6. **Model Architecture:**\n"
    "- Build at least two hidden layers with BatchNormalization and Dropout after each\n"
    "- Set output units = number of target_columns for multilabel/multiclass, else 1\n"
    "- Choose depth & width by data shape: shallow/narrow for small datasets, deeper/wider for large datasets, scale units ≈ min(features×2,1024)\n"
    "- Leverage provided `examples` but adjust architecture based on dataset size, feature count, and target count\n"
    "- **Architectural Guidelines:**\n"
    "  - **Choose by data size:**\n"
    "    • If `n_samples < 10000` or `n_features < 100`:\n"
    "        – Build **two** Dense layers of sizes:\n"
    "            [min(n_features*2, 128), min(n_features, 64)]\n"
    "        – **No** BatchNormalization; Dropout ≤ 0.3\n"
    "    • Else:\n"
    "        – Build **2–4** Dense layers of sizes:\n"
    "            [min(n_features*i, 1024) for i in (2, 1, 0.5, 0.25)] (drop any <16 units)\n"
    "        – After each: BatchNormalization() + Dropout(≤0.4)\n"
    "\n"
    "***For all hidden **Dense** layers (except the final output), use ReLU activation***\n"
    "  - **Task subtype → head, loss, batch & metrics:**\n"
    "    **(Note: activation applies only to the final/output layer)**\n"
    "    * **binary-classification:**\n"
    "       – activation=sigmoid, loss=binary_crossentropy\n"
    "       – batch_size=64–256, metrics=['accuracy', tf.keras.metrics.AUC(), tfa.metrics.MatthewsCorrelationCoefficient()]\n"
    "    * **multiclass-classification (MAP@N):**\n"
    "       – activation=softmax, loss=sparse_categorical_crossentropy\n"
    "       – batch_size=32–128\n"
    "       – dynamically compute top_k as: \n"
    "       – num_classes = len(np.unique(y_enc)) if isinstance(y_enc, (list, np.ndarray)) else 3\n"
    "       – top_k = min(num_classes, 5)\n"
    "       – metrics = ['accuracy', tf.keras.metrics.SparseTopKCategoricalAccuracy(k=top_k, name=f'top_{top_k}_accuracy')] # use sparse version for integer labels  \n"
    "       – at inference: take the top-`top_k` softmax probabilities for submission\n"
    "    * **multi-label-classification:**\n"
    "       – activation=sigmoid, loss=binary_crossentropy\n"
    "       – batch_size=64–256, metrics=['accuracy',tf.keras.metrics.Precision(), tf.keras.metrics.Recall(), tfa.metrics.F1Score(num_classes=n_classes)]\n"
    "    * **regression:**\n"
    "       – activation=linear, loss=mean_squared_error\n"
```

```
        "       – batch_size=32–256,
metrics=[tf.keras.metrics.RootMeanSquaredError(name='rmse'),
tf.keras.metrics.MeanAbsoluteError(name='mae')]\n"
        "    * **time-series forecasting:**\n"
        "       – use chronological split\n"
        "       – **model:** stack LSTM layers (their internal activations are tanh + sigmoid
gates), **then** any Dense head\n"
        "       – activation=linear, loss=mean_squared_error\n"
        "       – epochs=10–50,
metrics=[tf.keras.metrics.RootMeanSquaredError(name='rmse')]\n"
        ")"
    "7. **Compile the model with the Adam optimizer and the chosen loss and metrics\n"
    "8. **Callbacks & Training**:\\n"
    "   start_time = time.time()  # capture before fit\\n"
    "   if y_val is not None:\\n"
    "       history = model.fit(X_train_proc, y_train, validation_data=(X_val_proc, y_val),
epochs=100, callbacks=callbacks, verbose=2)\\n"
    "   else:\\n"
    "       history = model.fit(X_train_proc, y_train, validation_split=0.2, epochs=100,
callbacks=callbacks, verbose=2)\\n"
    "   duration = time.time() - start_time  # compute after fit\\n"
    "9. **Evaluation & Logging**:\\n\"  \n"
    "   Don't use tensorflow_addons, it is no longer supported use more recent ways to
record metrics"
    "   Turn on the verbose and save the training and validtion accuracy and log of the last
epoch in a json file (results.json). It will have the following keys: {training_accuracy,
training_loss,validation_accuracy and validation_loss}\n"
    "   with open('results.json','w') as f: json.dump(results,f)\\n\"  \n"
    "# Infer id_col & target_columns from submission_example header\n\
    if any(not col.replace('.','').isdigit() for col in target_columns) or len(target_columns) >
1:\n\
        competition_problem_subtype = \"multi-label-classification\"\n\
    \n\
    10. **Prediction & Submission**:\n\
    raw_preds = model.predict(X_test_proc)\n\
    if competition_problem_subtype == \"multi-label-classification\": final = (raw_preds >
0.5).astype(int)\n\
    elif competition_problem_subtype in [\"multiclass\", \"multiclass-classification\"]: idxs =
raw_preds.argmax(axis=1); final = le.inverse_transform(idxs)\n\
    elif competition_problem_subtype == \"binary-classification\":\n\
        probs = raw_preds[:,1] if raw_preds.ndim==2 and raw_preds.shape[1]==2 else
raw_preds.flatten()\n\
        final = (probs > 0.5).astype(int)\n\
```

```
            elif competition_problem_subtype in
[\"continuous-regression\",\"quantile-regression\",\"multi-output
regression\",\"missing-value-imputation\"]:\n\
                final = raw_preds\n\
                if np.all(final >= 0): final = np.expm1(np.clip(final, a_min=None, a_max=20))\n\
            else: final = raw_preds\n\
            \n\
            # ensure 2D\n\
            if final.ndim == 1: final = final.reshape(-1,1)\n\
            submission = pd.DataFrame(final, columns=target_columns)\n\
            submission.insert(0, id_col, test_ids.reset_index(drop=True))\n\
            submission.to_csv('submission_result.csv', index=False)\n"

        ),
        "parameters": {
            "type": "object",
            "additionalProperties": False,
            "properties": {
                "competition_problem_description": {
                    "type": "string",
                    "description": "Dense competition description giving the core goal."
                },
                "competition_problem_subtype": {
                    "type": "string",
                    "enum": [
                        "binary-classification",
                        "multiclass-classification",
                        "multi-label-classification",
                        "time-series-forecasting",
                        "continuous-regression",
                        "quantile-regression",
                        "multi-output-regression",
                        "ordinal-regression",
                        "missing-value-imputation"
                    ],
                    "description": "Rely on this to choose splits, loss, activation, etc."
                },
                "dataset_metadata": {
                    "type": "string",
                    "description": "Full NLP explanation of the dataset, the columns that need to be
predicted and the training files provided"
                },
                "data_profiles": {
                    "type": "array",
```

```
    "description": "One entry per file after compaction",
    "items": {
      "type": "object",
      "required": ["file_name", "shape","targets"],
      "properties": {
        "file_name": { "type": "string" },

        "shape": {
          "type": "object",
          "required": ["rows", "cols"],
          "properties": {
            "rows": { "type": "integer" },
            "cols": { "type": "integer" }
          },
          "additionalProperties": False
        },

        "targets": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "additionalProperties": False
    }
  },
  "files_preprocessing_instructions": {
    "type": "string",
    "description": "Instructions for how to preprocess the raw files."
  },
  "submission_example": {
    "type": "array",
    "minItems": 1,
    "items": {
      "type": "object",
      "properties": {
      "column_name": { "type": ["string", "number"] },
      "value":      { "type": ["number", "string", "boolean", "null"] }
      },
      "required": ["column_name", "value"],
      "additionalProperties": False
    }
  },
  "files_list": {
    "type": "array",
```

```
                "items": {"type": "string"},
                "description": " list of all files included in the competition, decide whether there are testing files and whether you need to split the training dataset"
            },
            "examples": {
                "type": "array",
                "description": "Retrieved preprocessing and code snippets from solutions of top similar competitions, rely on them ",
                "items": {
                    "type": "object",
                    "additionalProperties": False,
                    "properties": {
                        "preprocessing_steps":  {
                            "type":"array",
                            "items":{"type":"string"}
                        },
                        "model_layers_code":    {"type":"string"}
                    },
                    "required":["preprocessing_steps","model_layers_code"]
                }
            },
            "notebook_code": {
                "type": "string",
                "description": "***The complete runnable Python notebook code wrapped in <Code>…</Code>."
            }
        },
        "required": [
            "competition_problem_description",
            "competition_problem_subtype",
            "dataset_metadata",
            "data_profiles",
            "files_preprocessing_instructions",
            "submission_example",
            "files_list",
            "examples",
            "notebook_code"
        ]
    },
    "strict": True
  }
]
```

```python
# Extracting the Keras model layer and fit block

extract_tools = [
  {
    "name": "extract_model_block",
    "type": "function",
    "description": (
        "Given the full notebook in `original_code`, return JSON with exactly two keys:\n"
        "- `model_block`: every line from the first `model =` up to the line **before** the `model.fit(...)` call;\n"
        "- `fit_call`: the **entire** `model.fit(...)` invocation (including all its arguments) as one string.\n"
         "Do not change anything—just extract those snippets."
    ),
    "parameters": {
      "type": "object",
      "properties": {
        "original_code": { "type": "string" },
        "model_block":   { "type": "string" }
      },
      "required": ["original_code", "model_block"],
      "additionalProperties": False
    }
  }
]
```

#Building a Keras Tuner model using the previously extracted, working Keras model

```
tuner_tools = [
   #Tuner
   {
      "name": "generate_tuner_schema",
      "type": "function",
      "description": (
         "***Return ONE JSON field `tuner_code` containing runnable Python wrapped in
<Code>…</Code>.***\n\n"
         "1. Choose profile → best tag overlap in `hyperparameter_bank` → `chosen`.\n\n"
         "• **Preserve** the definitions and usages of:\n"
         "  ```python\n"
         "  early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)\n"
         "  checkpoint     = ModelCheckpoint('best_model.h5', monitor='val_loss',
save_best_only=True)\n"
         "  ```\n"
         "  and plug these same variables into both `tuner.search(...)` and the final
`model.fit(...)`.\n\n"
         "# preserve input dim\n"
         "n_features = X_train_proc.shape[1]\n\n"
         "2. HyperModel\n```python\n"
         "import keras_tuner as kt\n"
         "from tensorflow.keras.layers import Input, Dense, Dropout\n"
         "from tensorflow.keras.models import Model\n\n"
         "class MyHyperModel(kt.HyperModel):\n"
         "    def build(self, hp):\n"
         "        layers = hp.Int('layers', **chosen['params']['layers'])\n"
         "        units  = hp.Int('units', **chosen['params']['units'])\n"
         "        act    = hp.Choice('activation', chosen['params']['activation']['values'])\n"
         "        drop   = hp.Float('dropout', **chosen['params']['dropout'])\n"
         "        opt    = hp.Choice('optimizer', chosen['params']['optimizer']['values'])\n"
         "        lr     = hp.Float('learning_rate', **chosen['params']['learning_rate'],
sampling='log')\n\n"
         "        inputs = Input(shape=(n_features,))\n"
         "        x = inputs\n"
         "        if 'lstm_units' in chosen['params']:\n"
         "           dense_units = hp.Int   ('dense_units',   **chosen['params']['dense_units'])\n"
         "           # time-series LSTM branch\n"
         "           for i in range(layers):\n"
         "               return_seq = (i < layers - 1)\n"
         "               x = LSTM(lstm_units, return_sequences=False)(x)\n"
```

```
"           x = Dense(dense_units, activation='relu')(x)\n"
"           x = Dropout(drop)(x)\n"
"       else:\n"
"           for _ in range(layers):\n"
"               x = Dense(units, activation=act)(x)\n"
"               x = Dropout(drop)(x)\n"
"       x = output_layer_original(x)  # keep orig head\n"
"       model = Model(inputs, x)\n"
"       model.compile(optimizer=opt, loss=original_loss, metrics=original_metrics)\n"
"       # stash for use in tuner.search\n"
"       return model\n"
"```  \n"
"*No extra `hp.*` calls.*\n\n"
"3. Replace `model.fit` with Bayesian only, **using the provided batch_size and epochs from the **hyperparameter_bank:\n"
"tuner = kt.BayesianOptimization(\n"
"    MyHyperModel(),\n"
"    objective='val_loss',\n"
"    max_trials=10,\n"
"    executions_per_trial=1,\n"
"    seed=42,\n"
"    overwrite=False,\n"
"    project_name='bayesian_tuner'\n"
")\n\n"
"if y_val is not None:\n"
"    tuner.search(\n"
"        X_train_proc, y_train,\n"
"        validation_data=(X_val_proc, y_val),\n"
"        batch_size=bs, epochs=ep,\n"
"        callbacks=[early_stopping, checkpoint]\n"
"    )\n"
"else:\n"
"    tuner.search(\n"
"        X_train_proc, y_train,\n"
"        validation_split=0.2,\n"
"        batch_size=bs, epochs=ep,\n"
"        callbacks=[early_stopping, checkpoint]\n"
"    )\n\n"
"model = tuner.hypermodel.build(\n"
"    tuner.get_best_hyperparameters(1)[0]\n"
")\n"
"```"
"4. Retrain `model` with the original callbacks and data, guarding against `None`:\n"
"```python\n"
```

```
            "if y_val is not None:\n"
            "    history = model.fit(\n"
            "        X_train_proc, y_train,\n"
            "        validation_data=(X_val_proc, y_val),\n"
            "        epochs=100, batch_size=bs,\n"
            "        callbacks=[early_stopping, checkpoint],\n"
            "        verbose=2\n"
            "    )\n"
            "else:\n"
            "    history = model.fit(\n"
            "        X_train_proc, y_train,\n"
            "        validation_split=0.2,\n"
            "        epochs=100, batch_size=bs,\n"
            "        callbacks=[early_stopping, checkpoint],\n"
            "        verbose=2\n"
            "    )\n"
            "```\n"
        ),
        "parameters": {
            "type": "object",
            "additionalProperties": False,
            "properties": {
                "competition_problem_description": {
                    "type": "string",
                    "description": "Full text description of the task."
                },
                "competition_problem_subtype": {
                    "type": "string",
                    "enum": [
                        "binary-classification",
                        "multiclass-classification",
                        "multi-label-classification",
                        "time-series-forecasting",
                        "continuous-regression",
                        "quantile-regression",
                        "multi-output-regression",
                        "ordinal-regression",
                        "missing-value-imputation"
                    ]
                },
                "model_block": {
                    "type": "string",
                    "description": "The code from the Keras version of `model =` through just before
`model.fit`, plus any batch_size/epochs definitions."
```

```
            },
            "hyperparameter_bank": {
               "type": "object",
               "description": "One selected profile from HYPERPARAMETER_BANK for this
task.",
               "additionalProperties": False,
               "properties": {
                  "params": {
                     "type": "object",
                     "additionalProperties": False,
                     "properties": {
                     "layers":       { "type": "integer", "minimum": 1, "maximum": 8, "multipleOf": 1 },
                     "units":        { "type": "integer", "minimum": 64, "maximum": 1024, "multipleOf":
64 },
                     "activation":   { "type": "string", "enum": ["relu", "tanh"] },
                     "dropout":      { "type": "number", "minimum": 0.0, "maximum": 0.5 },
                     "optimizer":    { "type": "string", "enum": ["adam"] },
                     "learning_rate":{ "type": "number", "minimum": 1e-5, "maximum": 1e-2 },
                     "batch_size":   { "type": "integer", "enum": [32,64,128,256,512,1024] },
                     "epochs":       { "type": "integer", "minimum": 10, "maximum": 200, "multipleOf":
10 },
                     "dense_units":  { "type": "integer", "minimum": 16, "maximum": 512,
"multipleOf": 16 }
                     }
                  },
               },
               "required": ["dropout","optimizer","learning_rate","batch_size","epochs",
"activation","layers","units"]
            },
            "tuner_choice": {
               "type": "string",
               "enum": ["gridsearch","bayesian","hyperband"],
               "description": "Which Keras Tuner class to use."
            },
            "tuner_code": {
               "type": "string",
               "description": "***The complete runnable Python notebook code wrapped in
<Code>…</Code> saved into the `tuner_code` JSON field."
            }
         },
         "required": [
            "competition_problem_description",
            "competition_problem_subtype",
            "model_block",
```

"hyperparameter_bank",
"tuner_choice",
"tuner_code"
]
}
}

]

````
# Merge the existing Keras outline with the new Keras Tuner model layer code

merge =
{
"role": "user",
        "content": (
            "Here is my full notebook:\n\n"
            "```python\n"
            f"{original_code}\n```\n\n"
            "And here is the new Keras-Tuner snippet (build, compile, search, retrain):\n\n"
            "```python\n"
            f"{tuner_snippet}\n```\n\n"
            "Please replace **only** the existing model-definition block—that is, **every line** \n"
            "from the first `model =` up to (but **not including**) the first `model.fit` call—with this Keras-Tuner snippet. \n"
            "**Keep** any variables it relies on (`n_features`, `n_classes`, `output_layer_original`, etc.) so it drops in cleanly, \n"
            "and **do not** touch imports, data loading, preprocessing, callbacks, logging, or the submission code. Return the full notebook text with only that block swapped out."
        )
    }
````