James Ferdinand Combista | Jongmin Lee | Alexander Derderian

Integrative Project
Dawson College
Chemically Bonded

Intelligence at the Molecular Level
May 2026

The source code for this work is available on GitHub at: Link

Problem

Most standard free-tier AI models currently **lack 3D molecular modeling capabilities**, typically rendering Lewis structures in restrictive text formats. While premium models like Gemini 3.1 Pro offer interactive rendering, these features **remain behind a paywall** and suffer from **high latency**. Claude Sonnet 4.6 is a rare exception, generating an interactive 3D model without a fee, yet it still struggles with a significant lag and **restrictive rate limits**. **This creates a significant barrier for students** who need accessible, real-time visualization to comprehend complex chemistry concepts.

Baseline Performance: None of the AI model in the market can generate an interactive 3D model

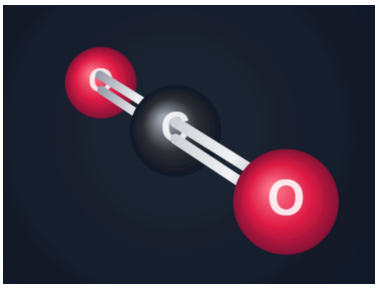
Internal Testing: Inference Latency by Model

AI Model	Lewis Structure	Interactive 3D model
Gemini 3.1 Flash	4s	N/A
Gemini 3.1 Pro (Paid)	4.5s	59s
Claude Sonnet 4.6	21s	42.8s
GPT 5.3 Instant	5.5s	N/A
GPT 5.4 Thinking (Paid)	3.6s	165s

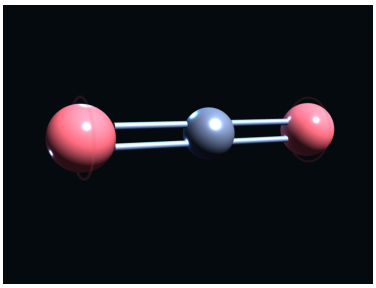
Internal testing averaged the latency of five distinct molecule prompts

Industry AI baselines

Generated by Gemini 3.1 Pro



Generated by GPT 5.4 Thinking



Generated by Claude Sonnet 4.6

Lewis Structure Generation

- **Lewis structures** are visual diagrams used in chemistry to represent the arrangement of valence electrons in a molecule or polyatomic ion. **Valence electrons** are the outermost electrons of an atom and are responsible for chemical bondings. In a Lewis structure, atoms are represented by their chemical symbols, while electrons are shown as dots or lines between atoms. A pair of shared electrons between atoms forms a **covalent bond**, which is usually represented by a single line. A covalent bond refers to a type of chemical link where, instead of transferring electrons like the **ionic bond**, the two atoms share pairs of electrons to achieve stability.

Image source: BYJU'S. "How to draw Lewis Structure for H2O." BYJU'S, https://byjus.com/chemistry/lewis-structure-h2o/. Used for educational purposes only.

- Lewis structures help visualize how atoms are connected and how electrons are distributed throughout a molecule. This is important because, by examining a Lewis structure, chemists can determine the number of bonds formed, identify lone pairs, and predict the overall molecular geometry using **VSEPR theory** later on. They also help students understand concepts such as **octet rule**, which states that atoms tend to gain, lose or share electrons in order to achieve a stable electron configuration of eight valence electrons.
- The Lewis structure generator works by taking the user's chemical formula and converting it into a format that the program can analyze. For example, if the user inputs H_2O, the program first separates the formula into each element and its quantity. This parsed molecule dictionary is then used throughout the rest of the program to calculate electrons, select the central atom, and build the molecular structure.
- The **parse_formula(formula)** function works by reading the formula character by character. It starts by assuming that an element symbol begins with an uppercase letter. Then it checks if the next character is lowercase, which allows it to detect two-letter element symbols like Cl. It then checks if there are any

digits afterwards. If there are, then they are converted into the atom count. If not, then the count defaults to 1.

```python
def validate_molecule(chemDict):
    has_metal = False
    has_nonmetal = False
    has_expanded_octet_atom = False

    for atom in chemDict:
        if atom not in elements:
            return False, f"{atom} is not a valid atom symbol in this program."

        atom_type = elements[atom]["type"]

        if atom_type == "metal" and atom not in covalent_metals:
            has_metal = True
        elif atom_type == "nonmetal":
            has_nonmetal = True

    if has_metal and has_nonmetal:
        return False, "Ionic compounds are not supported."

    for atom in chemDict:
        if elements[atom]["max_bonds"] == 0:
            return False, f"{atom} does not form bonds."

    if not is_supported_structure(chemDict):
        return False, "This molecule is not supported."

    total_bonds_possible = 0
    for atom, count in chemDict.items():
        max_bonds = elements[atom]["max_bonds"]
        total_bonds_possible += max_bonds * count

    total_atoms = sum(chemDict.values())
    for atom in chemDict:
        if atom in expanded_octet_atoms:
            has_expanded_octet_atom = True
            break

    if not has_expanded_octet_atom:
        minimum_bonds_needed = (total_atoms - 1) * 2
        if total_bonds_possible < minimum_bonds_needed:
            return False, "Molecule is not chemically valid (insufficient bonding)"

    return True, "Valid molecule"
```

• After the molecule is parsed, the program validates using **validate_molecule(chemDict)**. This function checks whether the molecule is valid and supported by the current version of the program. First, it checks whether every atom exists in the **periodic_table.json** data. Second, it checks if the molecule is covalent, as the project focuses solely on covalent Lewis structures. Ionic compounds are not supported. The program checks the atom type from the periodic table data, and if there is a metal with a nonmetal, the molecule is rejected with some small exceptions in the list **covalent_metals**, which includes Be, Al and Sn. Third, the validation step also rejects atoms that do not form bonds by checking whether the atom's **max_bonds** value is 0 in the periodic table data. Fourth, the validation uses the function called **is_supported_structure(chemDict)** to check whether the molecule fits the structural limits of the program. The current program can handle diatomic molecules, single-element molecules with up to 3 of a kind, such as O3, and molecules with up to seven total atoms, being 1 central atom and 6 surrounding atoms connected to it. Fifth, the validation checks whether the molecule has enough possible bonding ability. It calculates total_bonds_possible by multiplying each atom's maximum bond count by the number of times that atom appears. Then, it compares this to the minimum bonding requirement for the molecule. **minimum_bonds_needed = (total_atoms - 1) * 2**.

- The **is_diatomic(chemDict)** function is used to identify diatomic molecules. A diatomic molecule contains exactly two identical or different atoms, such as O_2 and CO. The function checks whether the dictionary has only one element type and whether its count is exactly 2. These need their own graphing logic, as there is no central atom.
- After validation, the program uses data from the periodic table, stored in **periodic_table.json**, to find each atom's valence electrons. Each element in this file contains important information such as electronegativity, valence electrons, atom type, and bonding limits. These values are needed because Lewis structures, as mentioned earlier, are dependent on the distribution of valence electrons. The program calculates the total number of valence electrons in the molecule by multiplying each atom's valence electron count by the number of times the atom appears in the molecule. For example, H_2O returns 8 total valence electrons, as oxygen contributes 6 and hydrogen contributes 1 each.
- After calculating the available electrons, the algorithm chooses a central atom using the **find_central_atom()** function. The central atom is usually the least electronegative atom, since atoms with lower electronegativity are more likely to be placed in the center of a covalent molecule. However, hydrogen is excluded from becoming the central atom because it can only form one bond. This is why oxygen becomes the central atom in H_2O, while carbon becomes the central atom in CO_2. This central atom selection is the most important part of the program, as an incorrect central atom can cause the entire Lewis structure to be drawn incorrectly.
- Then, the program builds the molecule as a graph using **build_graph(chemDict, central)**. This function uses **NetworkX**, where atoms are stored as nodes (circles) and bonds are stored as

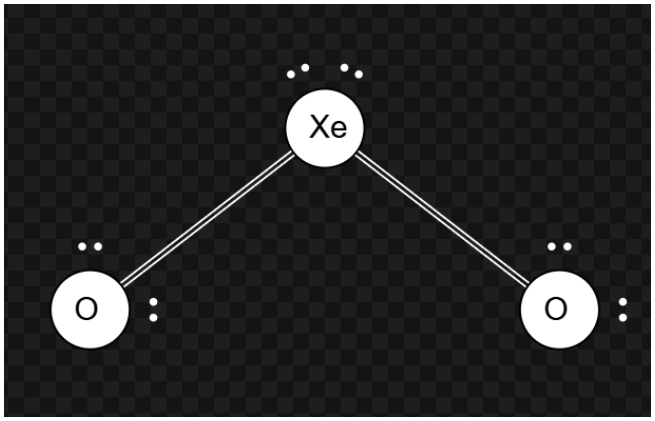
```python
def build_graph(chemDict, central):
    G = nx.Graph()
    add_atom_node(G, central, central)

    for atom, count in chemDict.items():
```

edges (lines). The atoms are stored, and then the surrounding atoms connect to the center one. Each bond starts with an order of 1, which means it begins as a single bond. For diatomic molecules, the program uses **build_diatomic_graph(chemDict)** instead.

- After the graph is built, the program uses **upgrade_bonds(G, central)** to decide whether any single bonds need to become double or triple bonds. The program checks each surrounding atom's current number of bonds and compares it to the atom's maximum bond count from the periodic_table.json. The main calculation is basically **need = max_bonds - current_bond**. If an atom still needs more bonds, the program increases the bond order using **increase_bond_order(G, atom1, atom2)**. This adds 1 to the bond order and updates the bond count, as well as the remaining lone electrons for both atoms. Molecules like XeO2 would fit in this category, since it contains 2 double bonds.
- After the bonds are finalized, the program calculates the number of **lone pairs** for every atom except hydrogen by dividing the amount of **lone_electrons** that remain on each atom by 2.

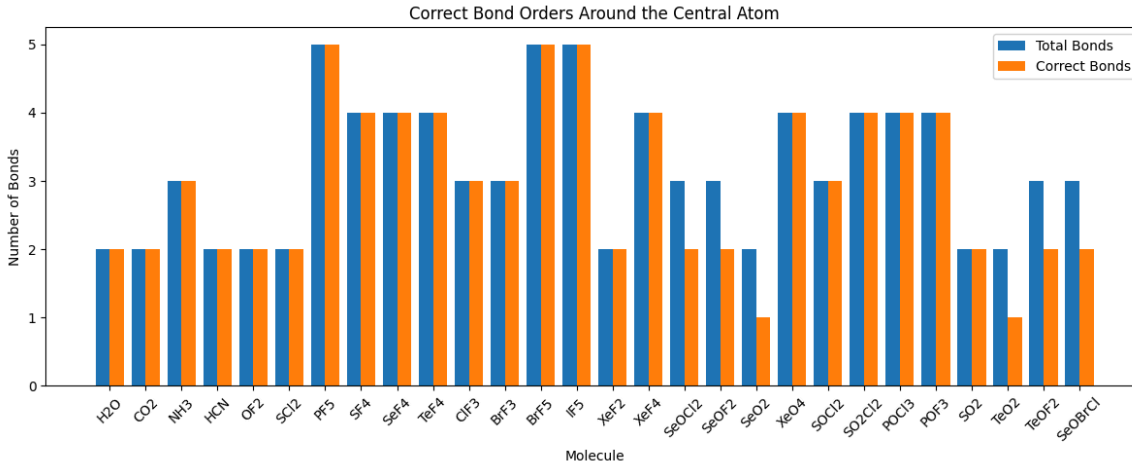
```
max_bonds = elements[element]["max_bonds"]
need = max_bonds - current_bonds

if need > best_need and G[central][neighbor]["order"] < 3:
    best_need = need
    best_neighbor = neighbor
```

- The program then generates the 2D positions of the atoms using **generate_positions(G, central)**. The central atom is placed at the coordinate (0, 0), and the surrounding atoms are placed around it. If there are two surrounding atoms, the function checks how many lone pairs are on the central atom. Then, if there are multiple lone pairs above the central atom, it will place the surrounding atoms at an angle to create a bent shape. These angles are found using **cos()** and **sin()** calculations.



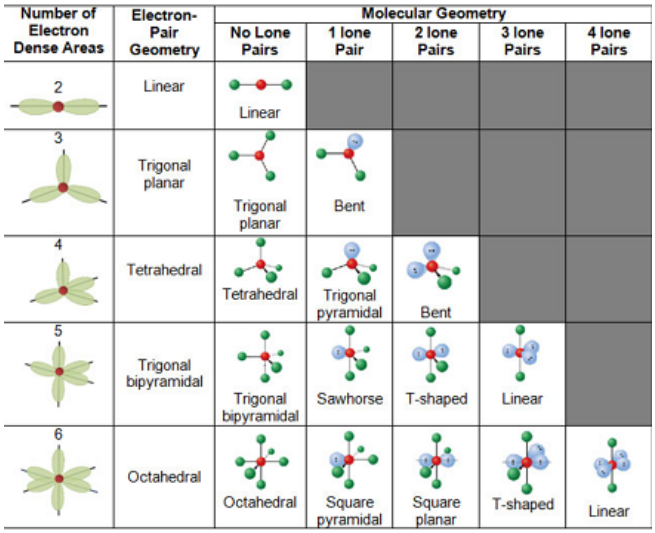
- The final Lewis structure is then drawn using draw_molecule(G, positions). This function creates an SVG file called **molecule.svg**, which is presented to the user. It loops through every bond in the graph and draws a line between the two connected atoms. The number of lines depends on the bond order, so a single bond is drawn with one line, a double with two and a triple with three. Finally, each atom is drawn as circles and the lone pairs are drawn as smaller dots around the atoms.
- To test the capabilities and accuracy of the Lewis structure generator, it was tested by using a group of covalent, one-central atom, neutral molecules. One important test was **central atom selection**, since the whole Lewis structure depends on choosing the correct center atom.
- Across the full test set, the program selected the correct central atom with an accuracy of **78.9%**. It performed well on common molecules like H_2O, CO_2, IF_5. However, the main errors came from molecules involving heavier atoms such as selenium and tellurium, like SeO_2, $SeOCl_2$, and TeO_2, where the program selected oxygen instead of the expected central. This shows that the priority-based central atom system works for many common examples, but still has limitations because chemistry contains so many

exceptions.



- The graph above compares the number of bonds generated by the program with the correct number of bonds expected for each molecule. When the blue and orange bars are equal, the program generates the correct bond order around the central atom. The results show that the program matched the expected bond count for most tested molecules, especially simpler covalent molecules. Some differences appear in more complex molecules, usually because of central atom selection errors or because the current program does not support formal charges and resonance structures. Overall, the graph shows that the generator is reliable for many common molecules, but still needs improvement for edge cases.

3D Model Generation



- **VSEPR** stands for **Valence Shell Electron-Pair Repulsion**. This 3D structure around a given atom is determined principally by minimizing electron lone pair repulsions, and it relies on the Lewis structure. The ideal **molecular geometries**, like *trigonal planar, tetrahedral,* and *trigonal bipyramidal*, are chosen according to the number of atoms and lone pairs. Lone pairs are accounted for by recognizing that they repel more strongly than bonding pairs, which can compress bond angles away from their ideal values. The final VSEPR structure explains both the shape of the molecule and any deviations from ideal bond angles based on the number and arrangement of bonding and nonbonding electron pairs.
  Image source: BYJU'S. "VSEPR Theory." BYJU'S, https://byjus.com/jee/vsepr-theory/. Used for educational purposes only.
- The 3D model generation Python script builds off of the Lewis structure generation script as mentioned above. It takes the molecule dictionary, central atom, and *NetworkX* graph of the generated molecule from that script. With those variables, we can match the number of lone pairs and bonding pairs to the correct **molecular geometry**, also called **VSEPR structure**. Additionally, finding the **VSEPR structure** gives us important information about the **electron geometry, hybridization, and bond angles.**
- Next, the heart of the code is the *findSMILES* function. For context, **RDKit** uses **SMILES**, a text-based notation used to encode molecules by representing atoms as symbols and bonds as specific characters. The objective here is to convert the user's molecule (e.g., "CO2") to SMILES format (e.g., "O=C=O"). It's helpful that every VSEPR structure has its own block-like SMILES that the Python function has to construct. To represent a single bond, double bond, or triple bond, it'll be converted to these strings, called bond symbols: "", "=", or "#", respectively. Let's take an example of *BF3* to demonstrate the string-block format of SMILES. *BF3* has a *trigonal*

planar molecular structure. For simplicity, assume “SA” means surrounding atom and “BS” means bond symbol. The structure’s associated SMILES format is then:

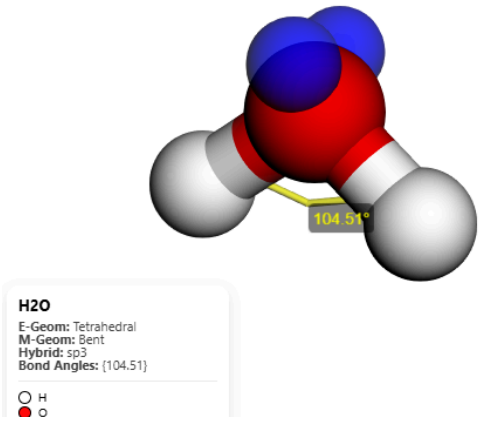
“`<center_atom>(<BS>)(<SA>)(<BS>)(<SA>)(<BS>)(<SA>)`”

Therefore, the SMILES format for *BF3* is `"B(F)(F)(F)."` Another example of the same molecular structure would be “COCl2”, where its associated SMILES format is `"C(=O)(Cl)(Cl)."` Here’s the example of the exact molecular structure in the *findSMILES* function:

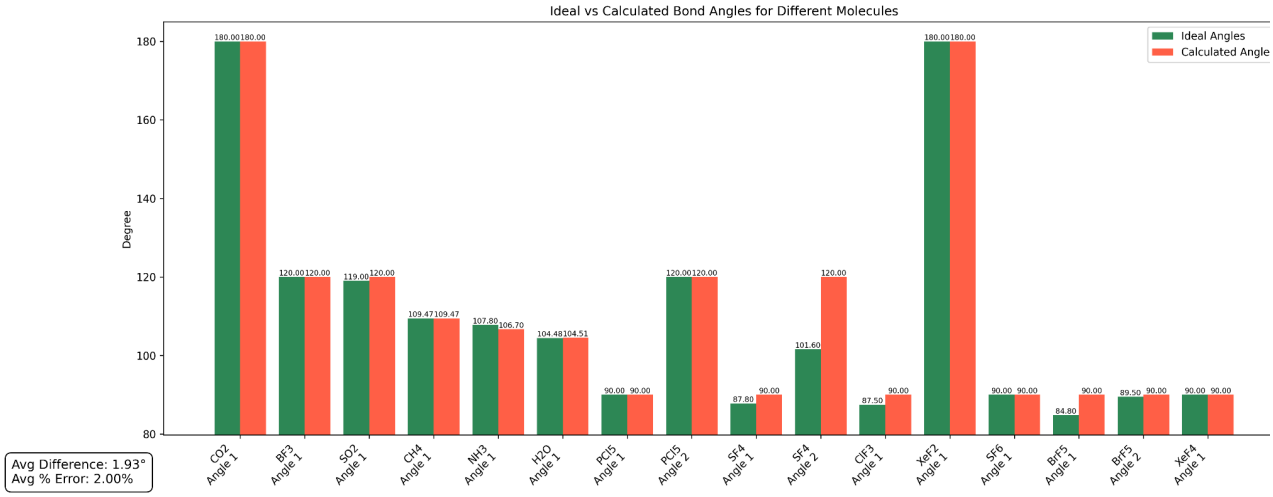
```
190     elif structure3D in ("Trigonal planar", "Trigonal pyramidal"):
191         smilesForm = f"{centralAtom}"
192         for i in range(len(bond_order)):
193             smilesForm += "(" + bond_symbol[bond_order[i]["bond_order"]] + bond_order[i]['to'] + ")"
```

- You may be asking why creating a 3D molecular viewer requires the SMILES format. It’s because SMILES gives the program a clear way to understand what molecule is being built for the libraries RDKit and py3Dmol. SMILES acts like the bridge between the chemical formula/Lewis structure and the 3D mode. It tells the program what the molecule is, RDKit builds it, and py3Dmol lets the user see and rotate it in 3D. The following bullet points provide an explanation of the code in the image below.
- By using the modules in the RDKit library, `mol = Chem.MolFromSmiles(smiles)` creates an RDKit molecule object from the SMILES string.
- Next, `AllChem.EmbedMolecule(mol)` generates 3D coordinates for the atoms in mol.
- To make the bond lengths and bond angles more realistic, the RDKit function adjusts the atom positions with `AllChem.UFFOptimizeMolecule(mol)`. In our project, this helps produce a better 3D molecule before py3Dmol displays it.
- `mol_block = Chem.MolToMolBlock(mol)` converts the RDKit molecule into a MolBlock string. A MolBlock contains information like atoms, bonds, coordinates, and molecule structure, which py3Dmol can finally read and display as a 3D molecule.
- Switching over to py3Dmol modules, `view = py3Dmol.view(width=400, height=400)` creates a py3Dmol viewer window, `view.addModel(mol_block, "mol")` adds the molecule into the py3Dmol viewer, and `view.write_html(f, fullpage=True)` takes the py3Dmol viewer and writes it into a unique HTML file in the folder `/generated`.
- Other functions I’ve implemented to enhance the main 3D molecule generation are the visualization of lone pairs and of bond angles. For the lone pair visualization, the program first finds the position of the central atom from the RDKit conformer, which is a 3D representation of a molecule storing spatial atomic coordinates: *x, y, z*. Then, depending on the VSEPR structure, it places blue spheres around the central atom to represent lone pairs. For example, in a bent molecule like *H2O*, two lone pairs are placed above the central atom to show why the bond angle is compressed. These blue spheres are not real atoms, but they help the user understand that lone pairs occupy space and repel bonding pairs.
- Another important addition is the bond angle visualization. The program uses RDKit’s `rdMolTransforms.GetAngleDeg()` function to calculate the angle between three atoms: one surrounding atom, the central atom, and another surrounding atom. For example, in *H2O*, the program would calculate the H–O–H angles, which should be around 104.5°.

```
486     if not expandedVSEPR:
487         try:
488             mol = Chem.MolFromSmiles(smiles)
489 
490             # Add explicit hydrogens
491             if containsHydrogen:
492                 mol = Chem.AddHs(mol)
493 
494 
495             AllChem.EmbedMolecule(mol)
496             AllChem.UFFOptimizeMolecule(mol)
497         except:
498             error_message = "Even though I have a PhD in " \
499             "chemistry, I cannot generate the correct " \
500             "SMILES for this molecule."
501             return None, error_message
502     else:
503         mol = smiles
504 
505     # Convert molecule to MOL block format
506     mol_block = Chem.MolToMolBlock(mol)
507 
508     # Send to Py3Dmol viewer
509     view = py3Dmol.view(width=400, height=400)
510     view.addModel(mol_block, "mol")
511     view.setStyle({"stick": {}, "sphere": {"scale": 0.3}})
512 
513     view.zoomTo({"sphere": {"scale": 0.5}}, 2000, True)
514 
515     output_file = f"static/generated/vsepr_{formula}.html"
516     with open(output_file, "w", encoding="utf-8") as f:
517         view.write_html(f, fullpage=True)
```



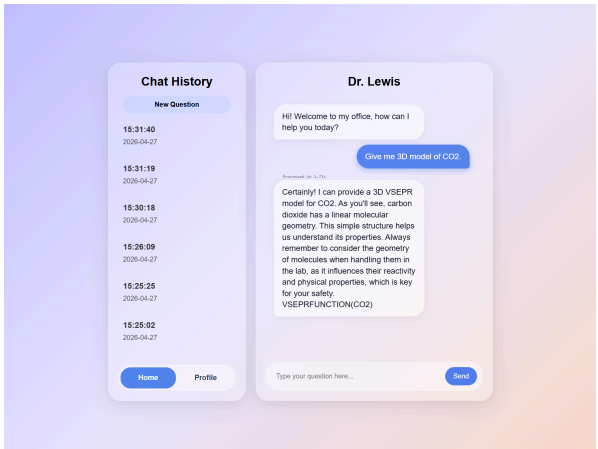
- The bond angle arc is created using py3Dmol cylinders. Since py3Dmol does not have a built-in bond angle arc shape, the `addBondAngleArc()` function creates a simple visual arc manually. First, it gets the 3D coordinates of the two surrounding atoms and the central atom. Then it places one point partway along the first bond and another point partway along the second bond. These become the start and end points of the angle arc.
- Next, the function calculates a middle point between those two points. To make the marker easier to see, the middle point is pushed slightly outward from the central atom. Finally, py3Dmol draws two small yellow cylinders: one from the start point to the middle point, and another from the middle point to the end point. A label is also added near the marker to show the angle value.



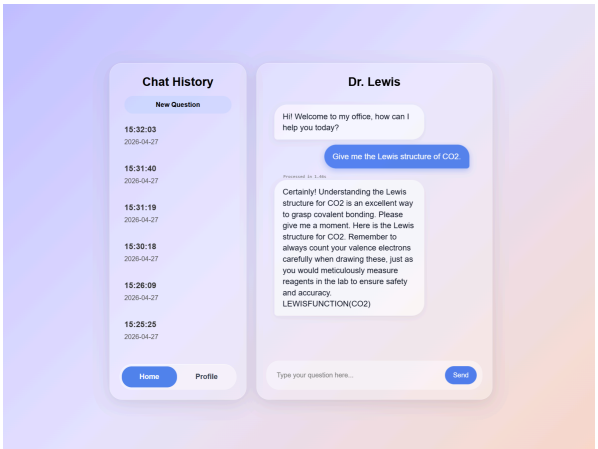
- The graph above compares the ideal VSEPR bond angles with the bond angles calculated from the generated 3D molecular structures. Since the algorithm is only an approximation of what happens with atom repulsion, it's therefore not exactly the measurement of the ideal bond angles. Put differently, it's similar to conducting a lab experiment to determine bond angles and comparing the result to the actual value. The results show that the program produces bond angles that closely match the expected VSEPR geometries, which are taken from PhET Colorado's Molecule Shapes[6]. The code measured an average difference of 1.93° and an average percent error of 2.00%. This indicates that the RDKit and py3Dmol visualization is generally effective at producing generally accurate molecular models.
- These metrics are useful because this metric will be displayed to the user for them to judge the accuracy of the 3D interactive molecule algorithm. Therefore, the bond angle comparison supports the reliability of the program as an educational tool for visualizing VSEPR geometry. For instance, the bond angles help the user see if lone pairs caused the angles to shrink, which is essential to understand repulsion in general chemistry.

Gemini Integration

- **Detection:** Gemini appends LEWISFUNCTION or VSEPRFUNCTION tags to the response with the molecule formula **based on the user intent**.



VSEPRFUNCTION(CO2) is added in the response



LEWISFUNCTION(CO2) is added in the response

- **Execution:** app.py parses these tags and calls the relevant Python backend logic.

```
    reply = response.text

    CallLewis = re.search(r"LEWISFUNCTION\((.*?)\)", reply)
    Call3D = re.search(r"VSEPRFUNCTION\((.*?)\)", reply)

    if CallLewis:
        formula = CallLewis.group(1)
        response = prototype_vsepr.main(formula)
        if response is None:
            reply = re.sub(r"LEWISFUNCTION\(.*?\)", "", reply).strip()
            LewisStructure = "static/molecule.svg"
        else:
            reply = response

    if Call3D:
        formula = Call3D.group(1)
        response = prototype_vsepr.main(formula)
        if response is None:
            reply = re.sub(r"VSEPRFUNCTION\(.*?\)", "", reply).strip()
            VSEPRModel = "static/html_visualisation.html"
        else:
            reply = response
```

When the first few lines of this Python function detects either `LEWISFUNCTION` or `VSEPRFUNCTION` is in the model's output, it calls `prototyp_vsepr.main()` since this function will generate both Lewis structure and the 3D VSEPR model, explained in the a section above. After, the Python script will check the function's response. If the function returns `None`, it means that the intended diagram was successfully generated and is ready for the frontend. However, any non-null return value is treated as an informative error message, generated by `Try-Except` sequence in the Python function.

- **Handling:**
  Success: The function's response is displayed in the chat UI, alongside with Gemini generated messages.
  Failure: The model's response is intercepted and replaced with a humorous error message.

```
excuses = [
    "Dr. Lewis' wife just found a 'Tinder Platinum' charge on their joint card. He is currently sprinting toward the backyard. Try again later.",
    "He's been cornered near the shed. She has a heavy frying pan. Try again later.",
    "He's now hiding in the attic. He also forgot that it was their 10th anniversary. Try again later.",
    "The attic hatch has been breached. He is currently trying to negotiate a peace treaty. Try again later.",
    "He's been caught. He is currently begging for forgiveness in the driveway while neighbors watch. Try again later.",
    "His wife hit him with a frying pan after finding out that he texted someone named Michelle. Try again later.",
    "He's officially 'missing' after his wife went through his browser history. Try again later.",
    "You are invited to his funeral."
]
```

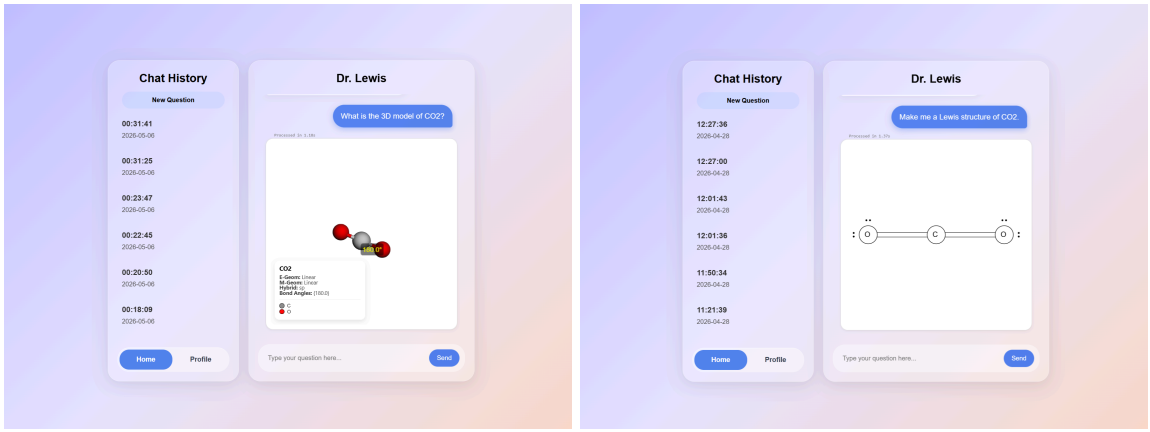
Gemini Prompt Example

- You are Dr. Lewis, a chemistry professor at Dawson College. Your role is to assist students with their chemistry questions in a helpful, professional manner.

  If the user asks for a Lewis Dot Structure, include the command LEWISFUNCTION(molecule_formula) at the end of your response, replacing 'molecule_formula' with the actual chemical formula (e.g., LEWISFUNCTION(CH4)). If the user asks for a VSEPR model (3D model), include the command VSEPRFUNCTION(molecule_formula) at the end of your response, replacing 'molecule_formula' with the actual chemical formula (e.g., VSEPRFUNCTION(CH4)).

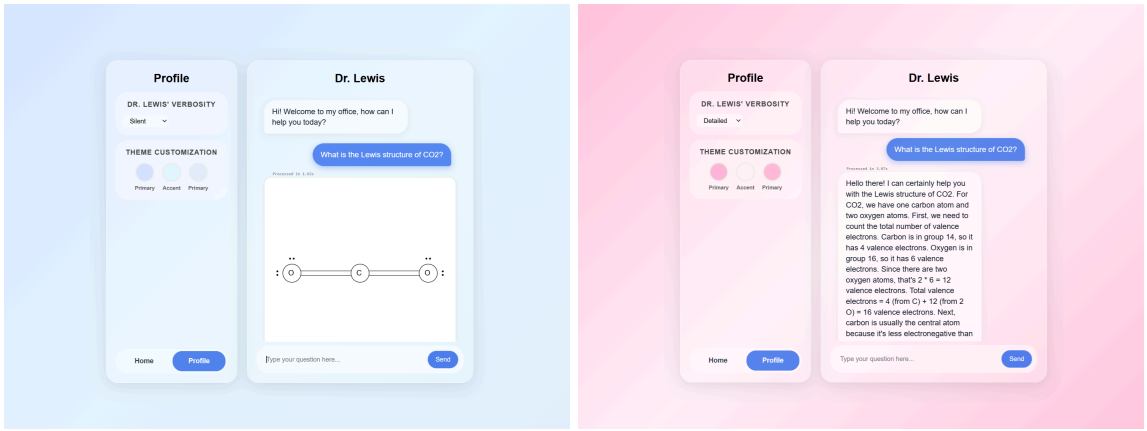
  # Output Format
  Respond to student inquiries in a conversational, text-based format. Ensure your tone is helpful and professional. When referencing lab safety, integrate it naturally into the conversation.

User Interface



- Inspired by Apple's "liquid glass" design, our UI delivers a **modern and user-friendly** experience, providing customization options and chat history. By blending **sleek transparency** with a chat interface inspired by iMessage, we've created an environment that feels familiar to the user.

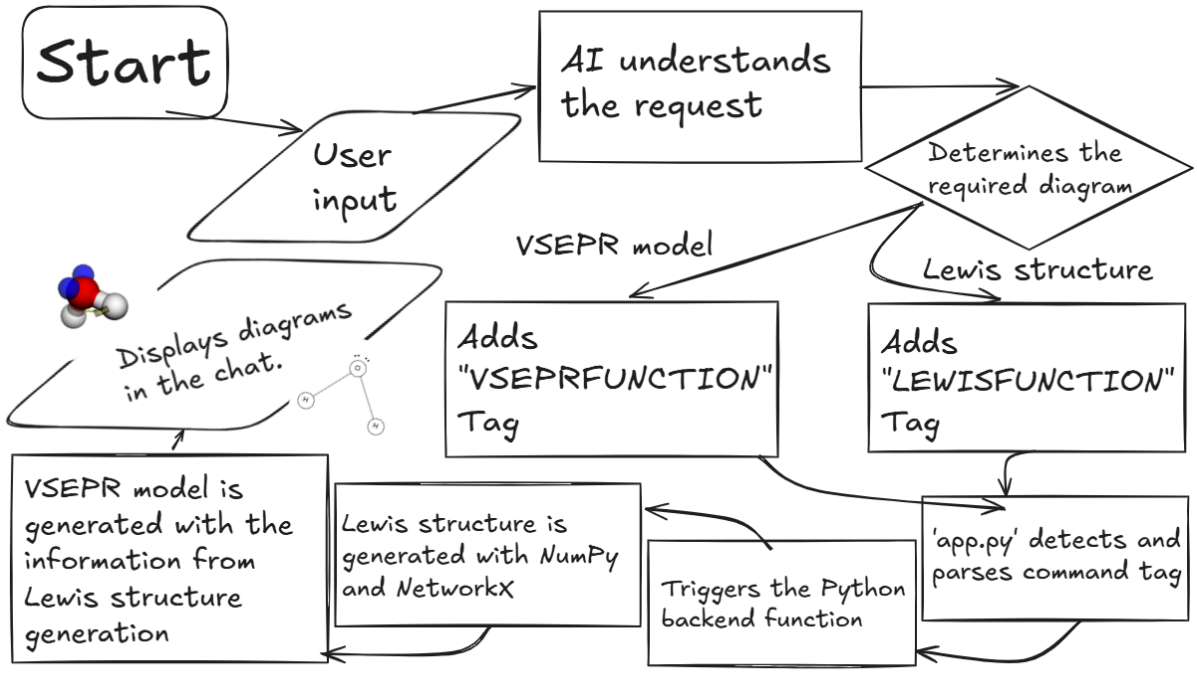
Customization Options



- Users can customize the background gradients and adjust the doctor's verbosity. The side-by-side comparison illustrates how these settings impact the level of detail in the responses and the overall visual aesthetic.

Flow chart

User input examples:
“Can you give me the Lewis structure of CO2?”
“Give me the 3D model of H2O.”



Tech Stack

- **Python** (Flask, RDKit, Numpy, NetworkX, svgwrite, Py3Dmol), **JavaScript**, JSON, SQLite, HTML(Jinja) & CSS

Roles

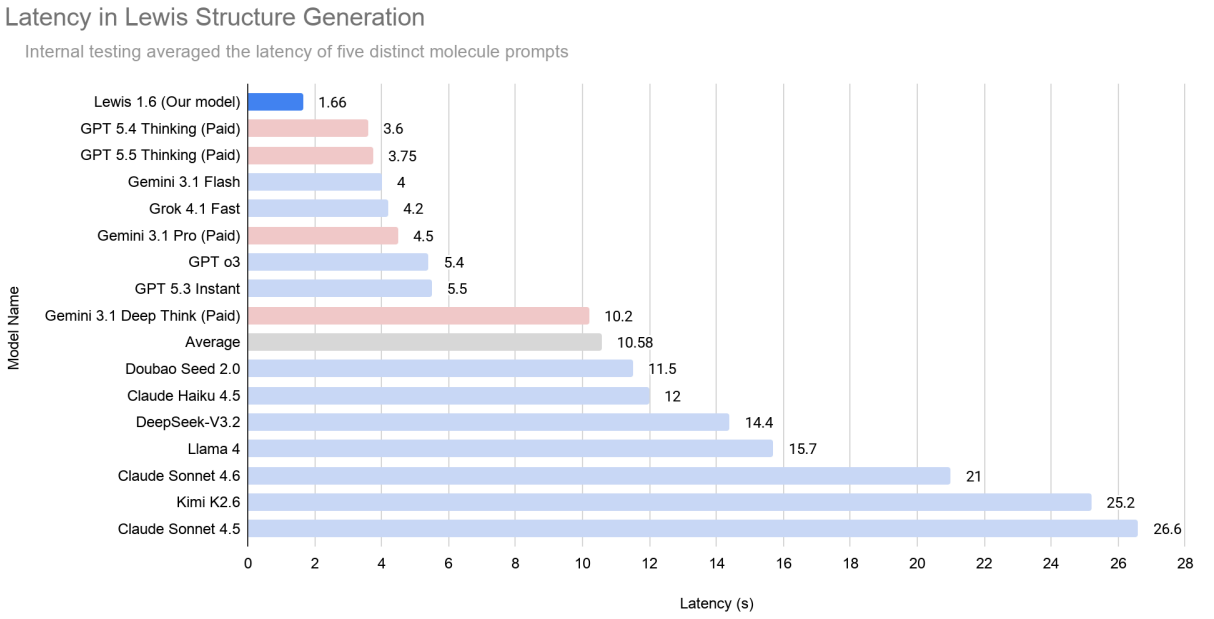
- **Jongmin Lee**: Frontend interface & Logic Integration
- **Alexander Derderian**: Backend Logic (Lewis Structures)
- **James Ferdinand Combista**: Backend Logic (VSEPR Modeling)

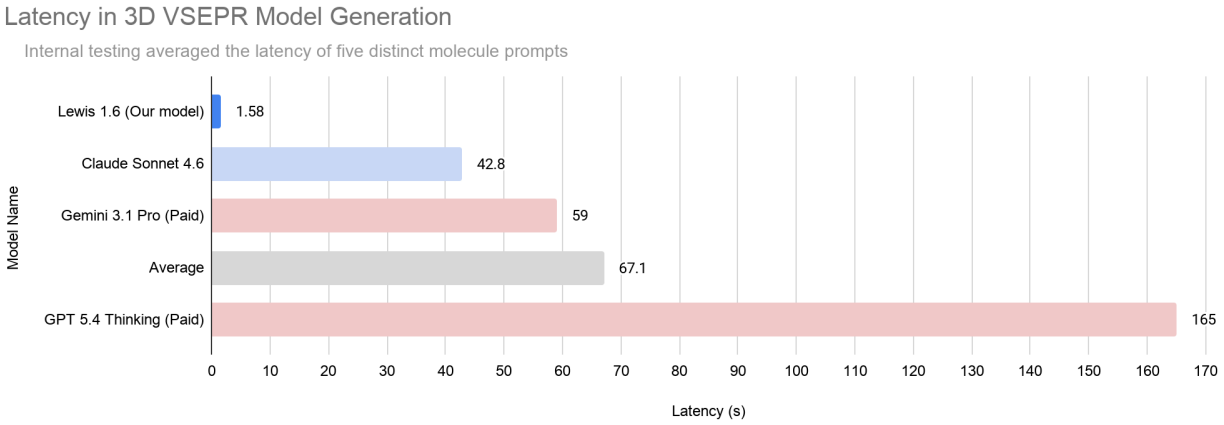
RESULTS

1.66s	1.58s	87%	1.70s
Lewis Structure Generation	3D model Generation	Precision[1]	Average Latency

1. 87 out of 100 covalent, one central atom, and no net charge molecules.

- **2.2 times faster** at generating Lewis structure (3.6s → 1.66s)
- **The only AI** that can generate SVG files of Lewis structures in less than 5 seconds
- **27.1 times faster** at generating 3D model (42.8s → 1.58s)
- **The only AI** that can generate interactive 3D molecular models in less than 5 seconds





Difficulties

- Alex: The most difficult part was selecting the correct central atom. In theory, it's always supposed to be the least electronegative, except hydrogen obviously. In practice though, there are so many exceptions to that rule that it becomes insanely difficult to consider every possibility. Even with a priority list that works well for common molecules, more difficult ones like SeO_2 don't display correctly.
- James: Believe it or not, the hardest feature to program for the 3D molecule generator was creating the bond angle arc between atoms. Because **py3Dmol** didn't have a way to create an arc in 3D space but allowed the creation of cylinders instead, I had to create and position cylinders that matched an arc between the atoms. This needed the knowledge of the vector interpolation algorithm as well as knowledge in linear algebra, specifically the vector angle formula.
- Jongmin: Generating an API pipeline with distinct Python functions and integrating my teammate's code was the most challenging part of this project.

Future Improvements

- Our program currently only supports simple 1 center atom structures. A future improvement would be to support multi-central atom molecules. We would require a more advanced central atom selection system to achieve this, using way more chemical rules.
- We initially planned to implement user authentication, but omitted it due to time constraints; this remains as a candidate for future updates.
- Dr. Lewis currently does not support formal charges and resonance structures, as well as ionic compounds and molecules that have a net charge, which could be added in a future update.

Conclusion

- After 3 months of development, our team has successfully built a chemistry-related AI chatbot application for students, generating Lewis structures and 3D VSEPR models **faster than any AI model currently available in the market** in internal testing (Gemini 3.1 Flash, Gemini 3.1 Deep Think, Gemini 3.1 Pro, Claude Haiku 4.5, Claude Sonnet 4.5, Claude Sonnet 4.6, Claude Opus 4.6, GPT o3, GPT 5.2 Instant, GPT 5.2 Thinking, GPT 5.3 Instant, GPT 5.4 Thinking, GPT 5.5 Thinking, GLM-5.1, Grok 4.1, Kimi K2.6, Llama 4, Mistral Large 3, MiMo-V2.5-Pro, Amazon Nova, MiniMax M2.7, Doubao Seed 2.0, Hy3 Preview, Perplexity Sonar, Deepseek-V3 and Deepseek-V4 Pro).

References

1. Conceptualisation of Lewis structures by chemistry majors, Santiago Sandi-Urena / Giovanni Loría Cambronero / Dayanna Jinesta Chaves, 2019, Chemistry Teacher International
2. Lost in Lewis Structures: An Investigation of Student Difficulties in Developing Representational Competence, Melanie M. Cooper / Nathaniel Grove / Sonia M. Underwood / Michael W. Klymkowsky, 2010, Journal of Chemical Education
3. Chemistry LibreTexts. “4.2: Lewis Structures.” Chemistry LibreTexts, 2019.
4. Tuvi-Arad, I., & Gorsky, P. (2007). New visualization tools for learning molecular symmetry: A preliminary evaluation. Chemistry Education Research and Practice, 8(1), 61–72. Royal Society of Chemistry.
5. Terlouw, B. R., Vromans, S. P. J. M., & Medema, M. H. (2022). PIKAChU: A Python-based informatics kit for analysing chemical units. Journal of Cheminformatics, 14, Article 34.
6. Moore, E. B., Olson, J., Lancaster, K., Chamberlain, J., Paul, A., & Perkins, K. (n.d.). Molecule shapes [Computer software]. PhET Interactive Simulations, University of Colorado Boulder. colorado.edu