# SQL Query Performance Tuning Best Practices (SELECT, INSERT, UPDATE, DELETE)

## SELECT Queries

### Speed and Efficiency

* **Retrieve only required columns (avoid** SELECT \***)** – Selecting all columns can force inefficient table scans and extra I/O. Unless every column is indexed, a SELECT \* query cannot use index-only access and will likely trigger full scans ([Jon Galloway - The real reason SELECT \* queries are bad: index coverage](https://weblogs.asp.net/jongalloway/the-real-reason-select-queries-are-bad-index-coverage" \l ":~:text=Image%3A%20SelectStarQueryPlan)). By specifying only needed columns, you enable the use of covering indexes and efficient index seeks (one test showed an index seek was 100× faster than the table scan caused by SELECT \* in SQL Server) ([Jon Galloway - The real reason SELECT \* queries are bad: index coverage](https://weblogs.asp.net/jongalloway/the-real-reason-select-queries-are-bad-index-coverage" \l ":~:text=Often%2C%20the%20bigger%20problem%20with,than%20the%20Clustered%20Index%20Scan)) ([Jon Galloway - The real reason SELECT \* queries are bad: index coverage](https://weblogs.asp.net/jongalloway/the-real-reason-select-queries-are-bad-index-coverage" \l ":~:text=Unless%20you%27ve%20indexed%20every%20single,extremely%20inefficient%29%20scan%20operations)).
* **Use WHERE filters to limit scanned rows** – Without a selective WHERE clause, the database must scan an entire table (a **full table scan**), which is very slow on large tables ([SQL index best practices for performance: 3 rules for better SQL indexes](https://www.cockroachlabs.com/blog/sql-performance-best-practices/" \l ":~:text=How%20much%20work%20does%20the,workload%20without%20using%20an%20index)). Always filter data as early as possible. For example, use WHERE instead of HAVING to apply conditions before aggregation whenever applicable ([Optimizing SQL Query Performance: A Comprehensive Guide | by Taran Kaur | Women in Technology | Medium](https://medium.com/womenintechnology/optimizing-sql-query-performance-a-comprehensive-guide-6cb72b9f52ef" \l ":~:text=Use%20WHERE%20Instead%20of%20HAVING)). Ensuring WHERE clauses are selective and indexed helps avoid reading unnecessary rows ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=14,and%20leverage%20the%20available%20indexes)).
* **Limit result sets if appropriate** – If you only need a subset of rows, use LIMIT, TOP, or FETCH to restrict the output. This prevents the overhead of generating and sending millions of rows that aren’t used ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=Consider%20using%20other%20methods%20like,GROUP%20BY%20if%20applicable)). It’s especially useful for pagination or reporting queries where showing the “top N” results suffices.
* **Ensure proper indexing for filters and joins** – Verify that columns used in WHERE conditions, join conditions, and ORDER BY/GROUP BY are supported by indexes. Indexes allow the database to efficiently locate and sort data instead of scanning everything ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=1,BY%20statements%20are%20indexed%20appropriately)). Without an index on a filter, the engine may load every row and then discard non-matching ones ([SQL index best practices for performance: 3 rules for better SQL indexes](https://www.cockroachlabs.com/blog/sql-performance-best-practices/" \l ":~:text=How%20much%20work%20does%20the,workload%20without%20using%20an%20index)). A good index will filter data efficiently, often eliminating the need for expensive in-memory sorting or interim scans.
* **Write SARGable conditions** – Make your filter conditions **Search-ARGument-Able**. Avoid wrapping indexed columns in functions or expressions (e.g. WHERE DATE(col) = '2025-01-01') because this can prevent index usage. Rewrite predicates to allow index seeks (for example, col >= '2025-01-01' AND col < '2025-01-02' instead of a function on col). This keeps the query optimizer able to use indexes for speed ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=14,and%20leverage%20the%20available%20indexes)).
* **Optimize JOIN usage and order in complex queries** – For queries joining many tables, the join order can impact performance. SQL optimizers will try to find an optimal join order, but they don’t try every combination ([sql server - Does Sql JOIN order affect performance? - Stack Overflow](https://stackoverflow.com/questions/16360860/does-sql-join-order-affect-performance" \l ":~:text=Although%20the%20join%20order%20is,of%20optimisation%20uses%20precious%20resources)). In very large join scenarios (e.g. 10+ tables with filters), a suboptimal join sequence can lead to huge intermediate results. It’s often best to join smaller, highly-filtered tables first so that later joins process fewer rows ([sql server - Does Sql JOIN order affect performance? - Stack Overflow](https://stackoverflow.com/questions/16360860/does-sql-join-order-affect-performance" \l ":~:text=Join%20order%20in%20SQL2008R2%20server,clauses%20applied%20against%20multiple%20tables)) ([sql server - Does Sql JOIN order affect performance? - Stack Overflow](https://stackoverflow.com/questions/16360860/does-sql-join-order-affect-performance" \l ":~:text=The%20trick%20is%20to%20set,data%20most%20through%20where%20clauses)). We have seen multi-join queries drop from 60+ seconds to under 1 second simply by rearranging join order to filter out data earlier ([sql server - Does Sql JOIN order affect performance? - Stack Overflow](https://stackoverflow.com/questions/16360860/does-sql-join-order-affect-performance" \l ":~:text=Although%20the%20join%20order%20is,of%20optimisation%20uses%20precious%20resources)). (Use caution: rely on the optimizer in most cases, but consider manual reordering or optimizer hints if you suspect a bad join plan in complex queries.)
* **Favor set-based operations over cursors or row-by-row logic** – Retrieving or processing data one row at a time (e.g. using cursors or application loops) is usually much slower than a single set-based query. Each loop iteration adds overhead. Whenever possible, let the SQL engine do the work in a single query or batch update. Cursors tend to be **slow and resource-intensive**, so avoid them unless absolutely necessary ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=operations)). For example, replace a cursor that aggregates values with a single SUM(...) GROUP BY query for drastic speed improvement.
* **Avoid unnecessary** DISTINCT **or sorting** – Using SELECT DISTINCT to eliminate duplicates, or sorting a large result, can be expensive in time and memory. Only use DISTINCT if you truly need unique results and cannot achieve it via schema design or EXISTS queries. If you need sorting, try to limit the data to sort (with WHERE/LIMIT) or ensure an index can provide the ordering to bypass a massive sort operation ([sql - Performance of order by on indexed vs non-indexed column - Stack Overflow](https://stackoverflow.com/questions/58040536/performance-of-order-by-on-indexed-vs-non-indexed-column" \l ":~:text=Yes%2C%20an%20index%20does%20improve,ORDER%20BY)). In summary, do not sort or de-duplicate more data than needed – it’s wasteful and slows down query execution.

### Memory and I/O Management

* **Leverage covering indexes to reduce I/O** – A covering index is an index that contains **all** the columns a query needs, so the engine can satisfy the query using only the index and avoid touching the main table. If you select only necessary columns (and they are indexed), the query can often be answered via an index-only scan (minimal I/O). For instance, removing an unneeded column from a SELECT allowed the query to use a nonclustered index alone, dropping the cost from 12.17 to 0.22 (in optimizer cost units) by eliminating expensive key lookups to the table ([Using Covering Indexes to Improve Query Performance - Simple Talk](https://www.red-gate.com/simple-talk/databases/sql-server/learn/using-covering-indexes-to-improve-query-performance/" \l ":~:text=Let%E2%80%99s%20begin%20by%20modifying%20our,with%20its%20actual%20execution%20plan)) ([Using Covering Indexes to Improve Query Performance - Simple Talk](https://www.red-gate.com/simple-talk/databases/sql-server/learn/using-covering-indexes-to-improve-query-performance/" \l ":~:text=The%20Estimated%20Operator%20Cost%20went,Operator%20Costs%20for%20each%20operation)). Thus, limiting columns not only speeds up execution but also greatly reduces disk reads and memory usage for caching data.
* **Use indexes to avoid large in-memory sorts/aggregations** – Operations like ORDER BY or GROUP BY on non-indexed columns require the database to sort or hash many rows, which consumes CPU and memory and may spill to disk. By indexing the columns used for sorting or grouping, you allow the database to retrieve data in sorted order or use index statistics for grouping, avoiding a costly full sort. Example: An ORDER BY on an indexed column can read data in index order, whereas on an unindexed column the server must load and sort all rows in memory ([sql - Performance of order by on indexed vs non-indexed column - Stack Overflow](https://stackoverflow.com/questions/58040536/performance-of-order-by-on-indexed-vs-non-indexed-column" \l ":~:text=Yes%2C%20an%20index%20does%20improve,ORDER%20BY)). Sorting is inherently expensive in CPU/memory, so let indexes do the work when possible (the larger the dataset, the more pronounced the benefit) ([sql - Performance of order by on indexed vs non-indexed column - Stack Overflow](https://stackoverflow.com/questions/58040536/performance-of-order-by-on-indexed-vs-non-indexed-column" \l ":~:text=Sorting%20is%20by%20design%20expensive,doing%20the%20sort%20in%20memory)) ([sql - Performance of order by on indexed vs non-indexed column - Stack Overflow](https://stackoverflow.com/questions/58040536/performance-of-order-by-on-indexed-vs-non-indexed-column" \l ":~:text=)).
* **Avoid cartesian products and overly broad joins** – A join missing a proper condition (or an unintended cross join) can produce a huge cartesian product of rows, consuming enormous memory and I/O. Always double-check join conditions to ensure you’re only combining intended rows. For analytical queries on many tables, consider interim temp tables or common table expressions to break down the problem and ensure you aren’t holding a gigantic intermediate result in memory. Keeping result sets and intermediate sets smaller (by filtering and correct joins) will prevent excessive memory usage and swapping.
* **Be mindful of query result size** – Pulling a very large result set can strain both server and client memory (the data has to be stored and transferred). If you need to retrieve millions of rows to an application, consider whether you can stream the results or process them in chunks. On the database side, fetching in smaller batches (using cursors or pagination queries) can reduce memory pressure. Also, ensure the client really needs all that data at once – often, such large results indicate work that could be pushed into the database (e.g., doing aggregation in SQL rather than fetching raw data to aggregate in application). Minimizing the data movement will improve overall performance and memory usage.
* **Use proper query plans and tools** – Examining the execution plan (using EXPLAIN) can reveal if the query uses disk-based operations (like tempdb spills in SQL Server or external merge sorts) due to memory-intensive steps. If you see hash joins or sorts spilling to disk, you may need to add indexes or rewrite the query. Regularly update your table statistics as well, since outdated stats can lead the optimizer to choose a plan that underestimates memory needs, causing unexpected I/O. Keeping stats current helps the optimizer allocate memory for joins/sorts more accurately ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=10,make%20better%20query%20execution%20plans)).

### Concurrency and Locking

* **Use appropriate isolation levels for heavy reads** – Long-running SELECT queries can coexist better with writing transactions by using a less restrictive isolation level. In systems like SQL Server, a default read will take shared locks that might block writers. To reduce lock contention, consider **READ UNCOMMITTED** (or the NOLOCK hint) for reporting queries that can tolerate dirty reads, or enable **Snapshot Isolation**/**READ COMMITTED SNAPSHOT** so that readers don’t block writers. This allows heavy SELECTs to run without locking data that other transactions might need, greatly improving concurrency (at the cost of allowing reading uncommitted data or using versioning). Always weigh data consistency vs. performance – use these techniques only when appropriate.
* **Keep transactions short** – Even for select queries, if you open an explicit transaction (e.g., doing some selects then later updates), don’t hold locks longer than necessary. Issue SELECTs outside of a transaction or commit promptly after the select if no longer needed. Long transactions (even read-only in serializable isolation) can retain locks or versioning information that pressures the system. The general rule is to do the minimum work required in a transaction and commit, so that locks (if any) are released quickly and the database can clean up row versions, freeing memory.
* **Avoid locking hints that escalate contention** – While one might be tempted to use locking hints like FOR UPDATE or HOLDLOCK on a SELECT to reserve data for subsequent updates, be cautious. These cause the select to take exclusive or update locks, blocking others. Instead, consider optimistic concurrency (select, then attempt update with a condition) or use a less blocking approach. In summary, for purely read queries, avoid any unnecessary locks; for read-modify workflows, try to narrow the locked range (e.g., lock just the necessary index rows) and keep the lock duration minimal to reduce impact on other queries.

## INSERT Queries

### Throughput and Speed

* **Batch multiple inserts in one statement** – Inserting many rows with a single INSERT ... VALUES that includes multiple tuples (or using bulk insert syntax) is significantly faster than one row per statement. This saves the database from parsing and executing numerous separate statements. According to MySQL’s documentation, using multi-row inserts can be **many times faster** than single-row inserts ([mysql - Which is faster: multiple single INSERTs or one multiple-row INSERT? - Stack Overflow](https://stackoverflow.com/questions/1793169/which-is-faster-multiple-single-inserts-or-one-multiple-row-insert" \l ":~:text=,row%20INSERT%20statements)). The reason is that network overhead, parsing, and per-statement setup are amortized over many rows instead of repeated every time. Always prefer batched inserts: for example, INSERT INTO table (col1,col2) VALUES (v1,v2), (v3,v4), ...; instead of looping one row at a time.
* **Group inserts into transactions (minimize autocommit)** – Each database commit has overhead (log flush to disk, etc.), so committing each row individually is very slow. It’s far more efficient to wrap multiple inserts in a single transaction, or batch commits after a number of rows. For instance, rather than 1000 single-row auto-committed inserts, do 1 transaction of 1000 inserts. This drastically reduces disk flush operations. In InnoDB/MySQL, the **slowest** method is one row per transaction, while a bunch of individual inserts in one explicit transaction is faster, and a single bulk insert in one transaction is fastest ([transaction - How does autocommit=off affects bulk inserts performance in mysql using innodb? - Database Administrators Stack Exchange](https://dba.stackexchange.com/questions/252445/how-does-autocommit-off-affects-bulk-inserts-performance-in-mysql-using-innodb" \l ":~:text=,is)). In short, **turn off autocommit** (or explicitly BEGIN…COMMIT) for bulk loads to get 10× or more throughput improvement by committing once after many inserts ([transaction - How does autocommit=off affects bulk inserts performance in mysql using innodb? - Database Administrators Stack Exchange](https://dba.stackexchange.com/questions/252445/how-does-autocommit-off-affects-bulk-inserts-performance-in-mysql-using-innodb" \l ":~:text=,is)).
* **Use bulk load tools for very large imports** – When you have to insert millions of rows, consider using database-specific bulk loading facilities (e.g. PostgreSQL COPY, SQL Server bcp/BULK INSERT, MySQL LOAD DATA). These are optimized for high-throughput data ingestion, often bypassing some of the overhead of SQL statements. Bulk loaders can use minimal logging or streaming protocols that are much faster than even batched INSERTs. Real-world tests in PostgreSQL have shown that COPY can outperform batched insert statements significantly for large data sets ([Multi-row insert vs transactional single row inserts](https://dba.stackexchange.com/questions/224989/multi-row-insert-vs-transactional-single-row-inserts" \l ":~:text=Multi,and%20it%20took%207%20seconds)). So, for one-time large imports or data warehouse loads, a bulk load will usually be the best choice.
* **Avoid excessive indexing and triggers during inserts** – Each insert must also update any indexes on the table and fire any triggers or foreign key checks. These extra operations can slow down bulk inserts dramatically. If you are loading data in bulk, consider disabling non-critical triggers and constraints beforehand, or dropping indexes (especially nonclustered indexes) and recreating them after the load ([SQL Update statement Performance Tips](https://www.sqlshack.com/sql-update-statement-performance-tips/" \l ":~:text=,clause%20in%20our%20update%20statements)). This trades some setup time for much faster insertion rate. For example, inserting into a table with zero indexes is much faster than into the same table with many indexes, where each row insert causes multiple index insertions ([mysql - Which is faster: multiple single INSERTs or one multiple-row INSERT? - Stack Overflow](https://stackoverflow.com/questions/1793169/which-is-faster-multiple-single-inserts-or-one-multiple-row-insert" \l ":~:text=,the%20numbers%20indicate%20approximate%20proportions)). (Always weigh this against the effort of rebuilding indexes and ensure data consistency if you bypass constraints.)
* **Use sequential access patterns if possible** – Inserting rows in an order matching the table’s clustering key (or primary key) can improve speed by reducing page splits or random I/O. For instance, inserting in ascending order of an indexed ID is typically faster than random inserts all over the index range. This isn’t always controllable, but if you have a choice (like when backfilling data), try to insert in key order. Similarly, inserting in smaller PK order chunks can help underlying storage (especially on clustered indexes) to remain compact and avoid fragmentation, boosting overall insert throughput.

### Resource Utilization (Memory, I/O)

* **Pick a reasonable batch size** – Very large insert batches in one transaction (e.g. millions of rows in one SQL statement or transaction) can consume a lot of memory (for buffering and for transaction logs) and can make the transaction log grow huge. It may also hold locks or latches for a long time. It’s often best to insert in chunks (say, 5k–50k rows per batch, depending on your system) rather than one giant transaction. This limits memory usage and log growth, while still benefiting from batching. Monitor your transaction log – if it’s becoming a bottleneck (growing very large or slowing down), reduce batch sizes and commit more frequently to free log space.
* **Monitor and optimize transaction log I/O** – Inserts (especially large batches) will generate log records. On systems with heavy insert throughput, ensure the log disk is fast (SSD/NVMe) and consider turning on optimizations like group commit. In SQL Server, using **minimal logging** for bulk insert (e.g., by enabling TABLOCK and using bulk-logged recovery model) can drastically reduce I/O. In PostgreSQL, batching inside one transaction will already optimize WAL (write-ahead log) usage. The key is to avoid small commits (which force frequent fsyncs) and also avoid unbounded transactions (which delay log reuse). Aim for a balance that maximizes sequential log writes but still allows periodic flush.
* **Watch out for foreign key cascades** – Inserting into a table with many foreign key relationships (especially ON INSERT triggers or cascading inserts) can spike CPU/memory as the database must verify or propagate each row. Ensure referenced tables have indexes on the foreign key columns to avoid full table scans during these checks. If you need to insert a lot of data with foreign keys, you might temporarily defer constraint checks (some databases allow deferring FK enforcement until commit) to speed up the bulk insert, then verify at the end. This can save memory and I/O during the insert process by not constantly scanning parent tables.

### Concurrency and Locking

* **Minimize lock contention on inserts** – Most modern engines (InnoDB, PostgreSQL, SQL Server) use row-level locks for inserts, so contention is low if inserts are into different rows. However, contention can arise if many transactions insert into the same table simultaneously. One contention point can be an **identity/autoincrement** generator or a hot index page (the end of an index if everyone inserts at the “end”). If you encounter insert bottlenecks due to locking or sequence generation, consider using multiple insertion threads each on separate key ranges, or use features like PostgreSQL’s COPY FREEZE to reduce locking overhead. Also, avoid holding locks by keeping insert transactions short – insert the data and commit promptly so other transactions aren’t waiting on your locks (especially if inserting into a table with exclusive constraints that lock a range).
* **Beware of table-level locks in certain engines or modes** – MySQL with MyISAM (an older engine) locks the entire table on inserts; similarly, using SQL Server’s BULK INSERT with a table lock hint will lock the table for the load. These can improve throughput for the insert itself but stop concurrent access. Use them only in off-hours or when concurrency isn’t a concern. Otherwise, stick to engines and methods that allow concurrent inserts (InnoDB, Postgres, etc. all allow it under most conditions). If you need high insert rate and concurrent reads, consider partitioning or sharding the table so that inserts go to a partition (or shard) that doesn’t lock other data. Reducing lock scope is key to keeping inserts scalable under concurrency.
* **Handle errors and rollbacks carefully** – Nothing hurts performance like inserting a million rows and then rolling back due to an error – it not only wastes time but also holds locks and occupies log space the whole time. To mitigate this, use smaller batches so that in case of error you only roll back the last batch, not the entire load. This practice reduces the time that locks are held (since each batch commits independently) and limits how long other transactions might be blocked waiting for a massive insert to rollback. In summary, batch your work so that any locking or rollback is bounded in scope.

## UPDATE Queries

### Speed and Efficiency

* **Always filter updates with a WHERE clause** – An UPDATE without a WHERE will attempt to modify every row of the table – a huge performance hit unless that’s truly what you intend. Ensure your UPDATE has a selective WHERE to target only the necessary rows ([SQL Update statement Performance Tips](https://www.sqlshack.com/sql-update-statement-performance-tips/" \l ":~:text=,we%20perform%20an%20update%20statement)). Even for broad changes, try to add conditions to avoid touching unaffected rows. This not only reduces work but also avoids unnecessary row locking.
* **Use indexes to locate rows to update** – Just like SELECTs, an update will benefit from an index on the columns in its WHERE clause. If no index supports the filter, the database may do a full table scan to find rows, significantly slowing the update. For example, if you are updating a subset of orders by customer\_id, make sure customer\_id is indexed so the engine can seek directly to those rows. The faster the engine can find the target rows, the faster the update can execute (the actual modification of each row is usually quick; the slow part is finding them without an index).
* **Avoid non-sargable predicates in the WHERE** – Similar to SELECTs, if you write UPDATE ... WHERE FUNCTION(column) = value or other non-sargable conditions, the DB cannot use an index on column. Rewrite conditions to be sargable (e.g., avoid WHERE YEAR(date) = 2025; instead use a range date >= '2025-01-01' AND date < '2026-01-01'). This ensures the update can seek or range-scan an index rather than scanning the whole table for matches.
* **Do not update columns unnecessarily** – Every updated row incurs overhead (writing to log, updating indexes, etc.), so avoid updating a row if the data isn’t actually changing. For instance, if your logic sets a status flag to “Y” but the row is already “Y”, skip it. A common pattern is adding ... WHERE new\_value <> old\_value to an update to only change truly different data. This minimizes work and especially reduces writes on already-up-to-date rows. It also helps avoid bumping row version timestamps or triggers when nothing actually changed.
* **Prefer set-based updates over row-by-row** – Updating many rows one at a time (in a loop or via a cursor) is usually much slower than a single SQL UPDATE that handles all rows at once. Set-based updates push the heavy work into the database’s optimized engine. For example, updating 100K rows with one UPDATE statement can be orders of magnitude faster than executing 100K individual update statements. The set-based approach optimizes I/O and logging by grouping operations. Only resort to row-by-row updates if each row requires complex, conditional logic that cannot be done in SQL (even then, consider if a single SQL CASE expression or MERGE statement can handle it).
* **Consider staged updates for complex logic** – If an update’s condition is complex or derived (e.g., “update these rows based on a subquery or calculation”), see if you can break it into simpler steps. Using a temporary table or common table expression (CTE) to hold keys of rows to update can sometimes be more efficient. For example, instead of a subquery for each row (WHERE id IN (SELECT ...) which might be re-evaluated repeatedly), do a prior SELECT to capture the target IDs, index that temp table, then join it in the UPDATE. This ensures the update operates on a readily available set of keys. This technique can speed up certain large or complex updates by simplifying the work done within the update statement.

### Managing Resources (Memory, I/O)

* **Update in manageable batches for large datasets** – Massive updates (affecting millions of rows) in one transaction can hog resources. Breaking the update into smaller batches (e.g. update 100k rows at a time, commit, then continue) often improves overall performance and reduces resource spikes. Batching limits how much the transaction log grows at once and frees memory periodically. In one experiment, updating ~4.8 million rows in one go took 37 seconds and 1.5 GB of log, whereas doing it in 1-million-row batches finished in 18 seconds with only 0.43 GB of log used ([Optimize Large SQL Server Inserts, Updates, Deletes with Batching - MSSQLTips.com](https://www.mssqltips.com/sqlservertip/5636/optimize-large-sql-server-insert-update-and-delete-processes-by-using-batches/" \l ":~:text=This%20time%20the%20query%20execution,was%20an%20improvement%20in%20time)). The batched approach nearly halved execution time and used ~70% less log space. This illustrates how chunking a huge update can make it more efficient by easing disk I/O and memory pressure.
* **Beware of lock escalation and tempdb usage** – On some systems (SQL Server in particular), an update that touches many rows will acquire many row locks which can trigger **lock escalation** to a table-level lock (often when >5,000 locks) ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=16)). This not only can block other queries, but it also indicates a very large operation. Batching, as mentioned, avoids hitting that threshold all at once. Also, if your update uses an execution plan that involves sorting, hashing, or spilling to tempdb (for example, updating with a join might spool data), monitor the tempdb or disk usage. You may need to add indexes or modify the query to be more efficient. If you must update a huge table entirely, doing it in smaller segments (by some key range or LIMIT/OFFSET in a loop) will mitigate tempdb and locking issues.
* **Disable or drop nonessential indexes and triggers** – When performing a bulk update, each updated row will also update all indexes on that table and fire any triggers. This can dramatically slow down the operation. If possible, disable triggers that aren’t needed for this one-time operation, and drop or disable indexes that you can rebuild afterward ([SQL Update statement Performance Tips](https://www.sqlshack.com/sql-update-statement-performance-tips/" \l ":~:text=,clause%20in%20our%20update%20statements)). Fewer indexes mean less work per row. For example, if a table has 5 secondary indexes, an update to a single column still requires up to 5 index updates. Removing some of those indexes (if they aren’t critical right now) could cut that overhead. Just remember to recreate any dropped indexes and ensure data integrity if you bypass triggers.
* **Check execution plans for inefficient operations** – Just as with SELECT, an update’s plan might reveal issues like a full scan or a suboptimal join. Use your database’s EXPLAIN or profile tools to see how the update executes. For instance, if you see a “Table Scan” in an update plan, that’s a red flag you might be missing an index on the filter. Or if you see a huge sort, maybe you’re updating with an ORDER BY unnecessarily (you usually don’t need to sort in an update). Optimizing the plan (through indexing or query rewrite) can reduce CPU and memory usage significantly. Also, updating large varbinary or JSON columns may cause heavy I/O – try to update such large fields only when necessary, and consider splitting them off to a separate operation if possible.

### Concurrency and Locking

* **Avoid long-running transactions for updates** – An update that affects many rows will hold exclusive locks on those rows (and possibly an exclusive table lock if escalated) until the transaction commits. To keep the rest of the system responsive, do not keep the transaction open any longer than needed. Commit as soon as the update batch is done. Long transactions not only block others but also increase the risk of deadlocks and can fill up lock memory. Keeping update transactions short and bite-sized is one of the most effective ways to reduce lock contention in a multi-user environment ([SQL Update statement Performance Tips](https://www.sqlshack.com/sql-update-statement-performance-tips/" \l ":~:text=,usage%20times%20for%20major%20updates)).
* **Perform major updates during off-peak hours** – If you need to update a very large portion of a table (say, adding a new column default or recalculating a field for all rows), schedule it during low-traffic periods. This minimizes the impact of locking and resource usage on other users ([SQL Update statement Performance Tips](https://www.sqlshack.com/sql-update-statement-performance-tips/" \l ":~:text=,usage%20times%20for%20major%20updates)). During off-peak times, it’s less likely that your update will collide with other transactions, and you can potentially allow a larger batch size or even do it in one go if acceptable. In a 24/7 system, you might still use small batches, but doing it when fewer users are active will reduce the chance anyone even notices slight slowdowns or locks.
* **Use row-versioning isolation if available** – Some databases provide MVCC (Multiversion Concurrency Control) or similar mechanisms so that readers aren’t blocked by writers. If using such a database (e.g., Oracle, PostgreSQL, or SQL Server with snapshot isolation), leverage it so that your big update doesn’t completely stop selects on the table. For example, in PostgreSQL an update doesn’t block selects; in SQL Server you can enable snapshot isolation so reads get the pre-update version of data. This doesn’t speed up the update per se, but it alleviates the impact on concurrency, effectively **reducing the lock contention** issue by allowing concurrent access. It can be a lifesaver for heavy updates on tables that still need to be read concurrently.
* **Consider alternative strategies for huge data changes** – If you need to alter most rows of a very large table, it might be worth considering a **bulk operation strategy** instead of a normal row-by-row update. One pattern is to create a new table (with the updated data or new schema), then swap it in place of the old table (renaming tables or using partition exchange if supported). This can be done with minimal locking (just schema locks at the end) and can avoid locking millions of rows altogether. While this is a complex approach, it underscores the idea that sometimes **refactoring the operation** (delete + insert new data, or using a staging table) can reduce locking impact and even improve speed compared to a gigantic in-place update.

## DELETE Queries

### Speed and Efficiency

* **Include a WHERE clause to narrow deletion** – As with updates, a DELETE without a WHERE will remove all rows in the table – essentially a full scan and massive operation. Only use it if you truly mean to wipe the table. Otherwise, always constrain which rows to delete. A selective WHERE allows the engine to use indexes to find the rows and avoid scanning everything. Even a simple condition like DELETE FROM Orders WHERE status='canceled' can use an index on status to target just those rows, instead of reading the entire Orders table.
* **Use indexes to locate rows to delete** – Make sure the filtering criteria for your delete (the WHERE clause) is indexed. Deleting 10,000 rows with an index to guide the search is much faster than deleting 10,000 rows found via a table scan. If you’re deleting based on a range or a key, the corresponding index will drastically reduce the amount of data the database engine must read. Conversely, if you delete based on a non-indexed column, the database will examine every row (poor performance). For large data purges, consider adding an index just for that purpose (if feasible) then drop it afterward. The index creation cost might be offset by the time saved in deletion.
* **Delete in smaller batches for large row counts** – Removing a huge number of rows in one command can be slow and also put great strain on the transaction log and locking (see below). It’s often faster to delete in chunks. For example, instead of DELETE FROM Logs WHERE log\_date < '2020-01-01' removing 50 million rows in one shot, you could loop, deleting 100k rows at a time (... LIMIT 100000 in MySQL, or using a TOP clause or a self-join trick in SQL Server). This approach allows the delete to commit periodically, freeing up resources. In practice, batch-deleting can significantly cut down runtime. It also gives other transactions breathing room between batches. A Stack Overflow example notes that deleting a million rows in one go can “destroy your transaction log,” recommending batch deletes instead ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=Yes%20you%20need%20to%20delete,will%20destroy%20your%20transaction%20log)).
* **Use** TRUNCATE **for full-table deletions** – If your goal is to delete all rows in a table, use TRUNCATE TABLE rather than a DELETE without a where clause. TRUNCATE is a DDL operation that quickly deallocates all data pages of the table, which is **much faster** and less resource-intensive than deleting row by row. It also typically uses minimal logging. As a trade-off, it usually requires higher privileges, and it will reset identity counters and cannot be used if there are foreign key references. But for bulk data removal (e.g., clearing a staging table), it’s the optimal choice (Brent Ozar quips: deleting all rows is “fast and easy – just do TRUNCATE TABLE” ([How to Delete Just Some Rows from a Really Big Table: Fast Ordered Deletes - Brent Ozar Unlimited®](https://www.brentozar.com/archive/2018/04/how-to-delete-just-some-rows-from-a-really-big-table/" \l ":~:text=Say%20you%E2%80%99ve%20got%20a%20table,percentage%20of%20them%2C%20say%205))). Always be absolutely sure you want everything gone, since truncate is irreversible and bypasses certain safety checks.
* **Consider partitioning or swapping for large deletes** – If you need to regularly purge old data (for instance, delete all rows older than X days), using partitioning can make this much faster. Rather than running a slow delete, you can partition the table by date and simply drop or truncate the oldest partition (which is a metadata operation, very fast). This achieves the deletion of those rows almost instantaneously. Another approach is to copy the rows you want to keep into a new table, then swap that in place of the old table (drop the old, rename the new). These techniques turn a heavy delete into quick metadata changes or bulk moves, greatly improving speed for large-scale deletions. They are advanced strategies, but worth it when dealing with billions of rows.

### Managing Impact on Resources

* **Beware of transaction log growth** – Deletes are fully logged operations (except in special bulk modes), meaning every deleted row is recorded in the transaction log. Deleting a huge number of rows in one transaction will bloat the log and could even fill it up or exceed I/O capacity. It’s not uncommon to see gigabytes of log generated for a large delete. Batching deletes (as mentioned) helps here – each small delete commits and frees log space. For example, deleting 1 million rows in one go will create a massive log entry, whereas 100 deletes of 10k rows will allow the log to truncate in between. As a rule, keep an eye on log file size during large deletions. If using SQL Server’s simple recovery, the log will truncate on each commit; in full recovery, ensure you take log backups or switch to bulk-logged mode if appropriate for the maintenance task.
* **Reduce locking and blocking** – A large delete will take **exclusive locks** on all affected rows and pages, and possibly escalate to a table lock if enough rows are affected at once ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=16)). While those locks are held, other transactions can be blocked (even reads, under many isolation levels, since a deleted row is effectively locked until the transaction completes). By deleting in smaller transactions, you avoid holding an exclusive lock on millions of rows at once. In SQL Server, for instance, deleting more than ~5000 rows in one transaction can escalate to a table lock, preventing even selects on that table until the delete commits ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=16)). Batching under that threshold (e.g. 1000 rows at a time) avoids lock escalation and keeps the locks more granular. This dramatically lessens the impact on other users – they might momentarily encounter a small locked portion, but not the whole table.
* **Handle foreign keys and dependencies carefully** – If the table you’re deleting from has foreign key relationships (child tables), deleting rows can also mean the database has to check or remove related rows. For example, deleting a customer might cascade delete their orders if ON DELETE CASCADE is set. Be mindful that such cascades multiply the work – the single delete statement could be effectively deleting from multiple tables. This can increase lock scope and resource use. In cases of mass deletion, you might disable foreign key constraints temporarily and manually handle the cleanup in the correct order (parent then child, or vice versa depending on need). Alternatively, break the operation: perhaps delete all child records first, commit, then delete parents, to avoid a single transaction doing too much. Managing referential integrity during large deletes is important to avoid long locks or excessive CPU from constraint checking.
* **Clean up and optimize after large deletes** – After removing a lot of data, the table (and its indexes) might be left with a lot of empty space. It can be beneficial to **rebuild indexes or run an optimize/ANALYZE** on the table afterward. This will recalculate statistics (so the optimizer knows the table is smaller now) and defragment space. Otherwise, you could have bloat that slows down subsequent queries. For example, in InnoDB, deleting a ton of rows might leave many pages half-full; an OPTIMIZE TABLE can reclaim space. In SQL Server, you might rebuild or reorganize indexes to consolidate free space. These maintenance steps ensure that the performance gains from having fewer rows are fully realized by the database. Also, if the deletion was so large that it outpaced the auto-update of statistics, manual update of stats will guide the query planner to better decisions (recognizing that the row count dropped significantly).

### Concurrency and Locking Considerations

* **Batch deletes to prevent long locks** – As reiterated, breaking deletes into chunks not only helps performance but also releases locks periodically. This is crucial for high-concurrency systems. Users can tolerate slight slowdowns, but not a table being frozen for minutes. Deleting say 1000 rows and committing will lock at most those 1000 at a time, allowing others to continue working on other parts of the table. It also avoids lock escalation to table-level in many databases. One user reported that deleting 1M+ rows in one shot “locked the entire table in exclusive mode” in SQL Server, blocking all other access until completion ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=16)). Their solution was to delete in batches of a few thousand to avoid that **massive interruption** ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=If%20you%20delete%20more%20than,avoid%20such%20a%20massive%20interruption)).
* **Use low-impact methods for archiving purges** – If your application can tolerate it, consider performing large deletes in a more “online” fashion: for example, instead of an explicit large DELETE transaction, you might mark rows as inactive (update a flag) quickly, then have a background job actually remove them in small chunks. The flag flip is fast and minimally locking, and the slow delete work happens asynchronously. This pattern can be safer concurrency-wise. Another approach is to leverage **Snapshot isolation** or similar if available, so that readers aren’t blocked by the deletions; they’ll see the old data until the transaction commits. This way, even if you have to delete a lot in one go, readers can continue (they read old versions). The writers (deleters) still acquire locks, but MVCC at least keeps read queries happy.
* **Coordinate with application locking if needed** – In some scenarios, you may want to take an application-level lock or maintenance mode when doing huge deletions (or updates). This isn’t a database feature per se, but if a deletion is so large that it will inevitably lock a table for a while, it might be better to pause certain application actions. For example, pause a queue or user activity that hits that table, perform the delete as quickly as possible, then resume. This avoids deadlock scenarios or timeouts for user transactions. It’s essentially performing the operation in isolation. While not always possible, for very heavy maintenance deletes, planned downtime or at least quiescing activity can ensure no concurrency issues – and thus no performance surprises.

**Sources:** The above best practices are drawn from real-world database performance experiences and recommendations across PostgreSQL, MySQL, and SQL Server. Key references include performance guides and Q&A discussions ([Jon Galloway - The real reason SELECT \* queries are bad: index coverage](https://weblogs.asp.net/jongalloway/the-real-reason-select-queries-are-bad-index-coverage" \l ":~:text=Often%2C%20the%20bigger%20problem%20with,than%20the%20Clustered%20Index%20Scan)) ([Jon Galloway - The real reason SELECT \* queries are bad: index coverage](https://weblogs.asp.net/jongalloway/the-real-reason-select-queries-are-bad-index-coverage" \l ":~:text=Unless%20you%27ve%20indexed%20every%20single,extremely%20inefficient%29%20scan%20operations)) ([sql server - Does Sql JOIN order affect performance? - Stack Overflow](https://stackoverflow.com/questions/16360860/does-sql-join-order-affect-performance" \l ":~:text=Join%20order%20in%20SQL2008R2%20server,clauses%20applied%20against%20multiple%20tables)) ([sql server - Does Sql JOIN order affect performance? - Stack Overflow](https://stackoverflow.com/questions/16360860/does-sql-join-order-affect-performance" \l ":~:text=The%20trick%20is%20to%20set,data%20most%20through%20where%20clauses)), database documentation on optimized inserts ([mysql - Which is faster: multiple single INSERTs or one multiple-row INSERT? - Stack Overflow](https://stackoverflow.com/questions/1793169/which-is-faster-multiple-single-inserts-or-one-multiple-row-insert" \l ":~:text=,row%20INSERT%20statements)) and indexing strategies ([sql - Performance of order by on indexed vs non-indexed column - Stack Overflow](https://stackoverflow.com/questions/58040536/performance-of-order-by-on-indexed-vs-non-indexed-column" \l ":~:text=Yes%2C%20an%20index%20does%20improve,ORDER%20BY)), as well as expert advice on batching write operations to avoid locking and log saturation ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=16)) ([Deleting 1 millions rows in SQL Server - Stack Overflow](https://stackoverflow.com/questions/24785439/deleting-1-millions-rows-in-sql-server" \l ":~:text=Yes%20you%20need%20to%20delete,will%20destroy%20your%20transaction%20log)). By following these guidelines – avoiding obvious pitfalls like SELECT \* and unfiltered updates, and using indexes, query rewrites, and batch techniques – you can achieve substantial gains in query speed and application throughput. Always remember to test changes in a safe environment, as the impact can vary based on data distribution and database engine specifics ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=14,and%20leverage%20the%20available%20indexes)) ([SQL Performance](https://www.linkedin.com/pulse/sql-performance-balwant-singh" \l ":~:text=15,using%20JOINs%20or%20other%20techniques)). The goal is to make queries **do less work** and **do it more intelligently**, which is the heart of SQL performance tuning.

---

Perfect, thanks! I’ll generate a complete list of prompt categories for detecting code issues in Python backend/server applications, focusing on application-level vulnerabilities and implementation faults. I’ll also include suggestions for future expansion—such as system-level misconfigurations—as a final section for your hackathon report. I’ll format everything into a clean checklist you can use to generate prompts in advance.

I’ll get back to you shortly with that organized list.

# Scanning Prompts for Python Application Vulnerabilities

## Authentication & Credential Issues

* **Hardcoded Passwords or Secrets**: Storing sensitive credentials (passwords, API keys, tokens) directly in code instead of using secure storage. This is dangerous because if the code is leaked, these secrets are immediately exposed ([Risks of Hardcoding Secrets in AI-Generated - Code](https://cycode.com/blog/the-risks-of-hardcoding-secrets-in-code-generated-by-language-learning-models/" \l ":~:text=Understanding%20Hardcoded%20Secrets)). Example Pattern: API keys assigned to variables or config values like PASSWORD = "admin123". LLM Detection: **High.** An LLM can easily spot common secret keywords (e.g. "password", "api\_key") and string literals, flagging hardcoded credentials in source code ([Risks of Hardcoding Secrets in AI-Generated - Code](https://cycode.com/blog/the-risks-of-hardcoding-secrets-in-code-generated-by-language-learning-models/" \l ":~:text=hardcoded%20secrets%2C%20the%20LLM%20might,it%20has%20observed%20during%20training)).
* **Insecure Password Storage (Plaintext or Weak Hashing)**: Using unhashed or weakly hashed passwords for user authentication. Passwords should **never** be stored in plaintext – proper hashing (with salt) is required ([Password Storage - OWASP Cheat Sheet Series](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html" \l ":~:text=Hashing%20and%20encryption%20can%20keep,should%20be%20hashed%2C%20NOT%20encrypted)). Using outdated algorithms like MD5 or SHA1 is also insecure, as these are fast to brute-force ([What is an insecure hash? | Tutorial & examples | Snyk Learn](https://learn.snyk.io/lesson/insecure-hash/" \l ":~:text=larger%20attack,that%20is%20a%20bad%20practice)). Example: Code that stores a user password directly in a database, or hashes with hashlib.md5() instead of a strong algorithm. LLM Detection: **High.** Clear indicators like md5( or literal password usage can be caught by pattern matching. An LLM can recognize the use of weak hash functions or absence of any hashing function for passwords and flag it as a vulnerability ([What is an insecure hash? | Tutorial & examples | Snyk Learn](https://learn.snyk.io/lesson/insecure-hash/" \l ":~:text=We%20should%20be%20aware%20of,that%20is%20a%20bad%20practice)).
* **Default or Weak Credentials**: The application uses default accounts or trivial passwords that were never changed (e.g. an admin account with password “admin” or “password123”). These weak credentials provide an easy entry point for attackers ([Pentester Guide: Weak or Default Credentials | Cobalt](https://www.cobalt.io/blog/weak-or-default-credentials" \l ":~:text=Weak%20or%20default%20credentials%20refer,an%20easy%20target%20for%20attackers)) ([Pentester Guide: Weak or Default Credentials | Cobalt](https://www.cobalt.io/blog/weak-or-default-credentials" \l ":~:text=Once%20these%20interfaces%20are%20identified%2C,to%20a%20full%20system%20compromise)). Example: An initialization script that creates a user admin/admin without requiring a reset. LLM Detection: **Moderate.** An LLM can identify obvious hardcoded defaults (like the string "admin" paired with "admin"), but subtle cases of weak passwords may require contextual judgment. It can flag known default combos and simple passwords by comparing against common credential patterns.

## Input Validation & Injection Flaws

* **SQL Injection (Unsafe Query Construction)**: Building SQL query strings by concatenating or formatting user input directly, allowing attackers to inject SQL code ([SQL Injection in Python | SecureFlag Security Knowledge Base](https://knowledge-base.secureflag.com/vulnerabilities/sql_injection/sql_injection_python.html" \l ":~:text=Since%20the%20SQL%20query%20is,and%20bypass%20the%20login%20mechanism)). Without input sanitization or parameterized queries, an attacker can modify query logic (e.g. ' OR 'a'='a' in a login query) to bypass authentication or exfiltrate data ([SQL Injection in Python | SecureFlag Security Knowledge Base](https://knowledge-base.secureflag.com/vulnerabilities/sql_injection/sql_injection_python.html" \l ":~:text=Since%20the%20SQL%20query%20is,and%20bypass%20the%20login%20mechanism)). Example: cursor.execute("SELECT \* FROM users WHERE name = '%s'" % user\_input) – if user\_input contains ' OR '1'='1', it manipulates the query. LLM Detection: **High.** The pattern of string concatenation or f-string with SQL keywords and user variables is recognizable. An LLM can infer risk when it sees raw queries with "%s" or {} placeholders filled by external input ([SQL Injection in Python | SecureFlag Security Knowledge Base](https://knowledge-base.secureflag.com/vulnerabilities/sql_injection/sql_injection_python.html" \l ":~:text=,%28username%2C%20password)), flagging them as potential SQL injection points.
* **OS Command Injection**: Passing unsanitized user input to system shell commands (via os.system, subprocess.Popen with shell=True, etc.), which allows execution of arbitrary OS commands ([Command injection in Python: examples and prevention | Snyk](https://snyk.io/blog/command-injection-python-prevention-examples/" \l ":~:text=What%20is%20command%20injection%3F)) ([Command injection in Python: examples and prevention | Snyk](https://snyk.io/blog/command-injection-python-prevention-examples/" \l ":~:text=Unfortunately%2C%20it%20is%20easy%20to,method)). If an application constructs command strings directly from user data, attackers can append malicious commands (e.g. ; rm -rf /) to that input. Example: os.system(f"ping -c 4 {user\_host}") – an attacker supplying user\_host = "example.com; cat /etc/passwd" would execute both ping and cat commands. LLM Detection: **High.** An LLM can spot functions like os.system or subprocess.run with suspicious f-strings or string concatenation that include user-provided variables ([Command injection in Python: examples and prevention | Snyk](https://snyk.io/blog/command-injection-python-prevention-examples/" \l ":~:text=headers%29%20to%20a%20system%20shell)). These are strong red flags for command injection, and the model can reason that any use of shell=True or direct shell command construction is dangerous.
* **Cross-Site Scripting (XSS)**: Injecting malicious scripts into web page output by failing to sanitize user input before including it in HTML. In Python web apps (Flask/Django), this often means rendering user-provided data without proper escaping, so the browser executes an attacker's <script> code ([What is cross-site scripting (XSS)? | Tutorial & examples | Snyk Learn](https://learn.snyk.io/lesson/xss/" \l ":~:text=Cross,distinguished%20from%20a%20legitimate%20script)). Example: A Flask route that reads a message parameter and returns f"<h1>{message}</h1>" directly. If message is "<script>alert('xss')</script>", it will execute in the victim’s browser. LLM Detection: **Moderate.** The LLM can detect obvious cases where HTML is constructed directly from inputs or when output encoding is turned off. It looks for patterns like inserting variables into HTML strings or use of functions (like Jinja’s Markupsafe disabling) that indicate raw output. While the model might not simulate a browser, it knows that any HTML generation with user input should be sanitized and can warn if it doesn’t see evidence of that (e.g., no escaping function).
* **Other Injection Flaws**: The category also includes path injection (e.g. using unsanitized file path input, leading to **path traversal** attacks) and script injection in formats like YAML or XML. Example: Using open(user\_filename) directly could allow ../../etc/passwd style paths. LLM Detection: **Moderate.** The LLM can notice user input being used in file or OS APIs (file open, XML parsers) without cleanup. It infers the risk (like ../ in paths) from context, although deeper file-system knowledge is external. Prompt-based detection can still flag the pattern of using user-supplied filenames or XML without safety measures as suspicious.

## Authorization & Access Control

* **Missing Authorization Checks (Horizontal Access / IDOR)**: Failure to enforce permission on resource access, enabling Insecure Direct Object References. This occurs when user-supplied identifiers (IDs) can be used to access objects without verifying ownership ([Finding and fixing insecure direct object references in Python | Snyk](https://snyk.io/blog/insecure-direct-object-references-python/" \l ":~:text=has%20no%20access%20controls%2C%20any,in%20the%20URL)). For example, an API endpoint /users/{user\_id} that returns user data but does not confirm that the authenticated user owns that user\_id is vulnerable – an attacker can simply change the ID and retrieve another user’s data ([Finding and fixing insecure direct object references in Python | Snyk](https://snyk.io/blog/insecure-direct-object-references-python/" \l ":~:text=has%20no%20access%20controls%2C%20any,in%20the%20URL)). LLM Detection: **Moderate.** An LLM can pick up on routes or functions that access data by an identifier with no accompanying authentication or user check. If it sees queries like get\_user\_by\_id(id) without any filter on the current user or guard clause, it infers a possible IDOR. It uses naming and context (e.g., function names like get\_user\_by\_id in an open endpoint) to flag missing access control.
* **Unprotected Admin/Privilege Functionality**: Sensitive actions or admin-only endpoints that don’t enforce role checks. For instance, an admin panel route that anyone can reach, or role-based logic implemented only on the UI side. OWASP notes that simply modifying a URL or parameter might grant unauthorized admin access if the server doesn’t verify the user’s role ([A01 Broken Access Control - OWASP Top 10:2021](https://owasp.org/Top10/A01_2021-Broken_Access_Control/" \l ":~:text=Scenario%20,access%20to%20the%20admin%20page)). Example: A Flask view @app.route('/admin') that does not check current\_user.is\_admin and allows any logged-in user (or even an unauthenticated request) to execute admin operations. LLM Detection: **Moderate.** The model can identify functions or routes with names like “admin” or critical actions (create/delete user, change roles) that lack any guard (no decorator like @login\_required or no role conditional in code). It will flag the absence of a permission check. However, some logic flaws (like incomplete conditionals) might be subtler – the LLM relies on known patterns (e.g., an if admin check) to decide if something is missing.
* **Over-reliance on Client-Side Access Control**: Code that assumes the frontend or UI will prevent unauthorized actions, without enforcing rules on the backend. For example, not validating a user’s privilege on a form submission because the admin GUI is hidden for normal users. Attackers can directly call the backend API and bypass the UI. Example: A delete user API that trusts a JavaScript front-end to restrict the button to admins only. LLM Detection: **Low.** This issue is about absence of server checks, which the LLM can catch if it explicitly sees no verification in code. But often, detecting a negative (nothing present) is harder. The LLM might need hints (comments or known framework usage) to realize that only client-side control exists. It can still warn when an endpoint performs sensitive actions without any authentication/authorization logic present, since that strongly implies missing server-side enforcement.

## Cryptographic Misuse

* **Weak or Broken Cryptographic Algorithms**: Use of outdated or insecure cryptography methods that undermine security. Common examples include using MD5 or SHA-1 for hashing passwords, or DES for encryption ([Cryptographic Failures: Understanding and Preventing Vulnerabilities | by Ajay Monga | Medium](https://medium.com/@ajay.monga73/cryptographic-failures-understanding-and-preventing-vulnerabilities-91c8b2c56854" \l ":~:text=Weak%20or%20Broken%20Algorithms%3A%20Using,or%20insecure%20algorithms%2C%20such%20as)) – all of which are considered cryptographically broken. Similarly, using a cipher in an insecure mode (like AES in ECB mode) or too short keys weaken protection. Example: Code that does hashlib.md5(password).hexdigest() for storing a password, or uses the Crypto.Cipher.DES module for encryption. LLM Detection: **High.** The model can recognize specific API calls or constants (e.g., "MD5" or "DES") known to be weak. It knows these algorithms are deprecated for security use ([Cryptographic Failures: Understanding and Preventing Vulnerabilities | by Ajay Monga | Medium](https://medium.com/@ajay.monga73/cryptographic-failures-understanding-and-preventing-vulnerabilities-91c8b2c56854" \l ":~:text=Weak%20or%20Broken%20Algorithms%3A%20Using,or%20insecure%20algorithms%2C%20such%20as)). Prompt analysis can flag them, suggesting more secure alternatives. The LLM’s training on security best practices lets it infer that such usage is a vulnerability.
* **Hardcoded Keys or Credentials in Crypto**: Embedding cryptographic keys, secrets, or salts in code, rather than deriving or securely storing them. This is akin to hardcoded passwords, but for keys – if an attacker can read the code, they instantly obtain the encryption keys ([Cryptographic Failures: Understanding and Preventing Vulnerabilities | by Ajay Monga | Medium](https://medium.com/@ajay.monga73/cryptographic-failures-understanding-and-preventing-vulnerabilities-91c8b2c56854" \l ":~:text=Hardcoded%20Keys%20and%20Secrets%3A%20Storing,directly%20in%20the%20source%20code)). Example: SECRET\_KEY = "mysecret12345" at the top of a Django settings file, or an API token for a payment gateway defined in plain text. LLM Detection: **High.** Just like with credentials, any long string constant or obvious key in code (especially if used in crypto functions) will stand out. The LLM can spot variables named KEY, SECRET, TOKEN, etc., assigned string literals ([Cryptographic Failures: Understanding and Preventing Vulnerabilities | by Ajay Monga | Medium](https://medium.com/@ajay.monga73/cryptographic-failures-understanding-and-preventing-vulnerabilities-91c8b2c56854" \l ":~:text=Hardcoded%20Keys%20and%20Secrets%3A%20Storing,directly%20in%20the%20source%20code)). It will flag these as hardcoded secrets. Even if the key looks random, the presence of such a literal used in encryption or signing is a strong indicator of a problem.
* **Insufficient Randomness (Predictable Values)**: Using non-cryptographic random generators or static seeds for security-sensitive operations (tokens, passwords, IVs). For instance, using Python’s random.random() or random.randint() for generating an API token – these are predictable since random is not cryptographically secure ([Insecure Randomness | OWASP Foundation](https://owasp.org/www-community/vulnerabilities/Insecure_Randomness" \l ":~:text=Standard%20pseudo,withstand%20cryptographic%20attacks)). Another example is reusing an initialization vector or seed, which can lead to pattern leaks ([Cryptographic Failures: Understanding and Preventing Vulnerabilities | by Ajay Monga | Medium](https://medium.com/@ajay.monga73/cryptographic-failures-understanding-and-preventing-vulnerabilities-91c8b2c56854" \l ":~:text=Insufficient%20Randomness)). Example: verification\_code = random.randint(100000, 999999) for a one-time password, or random.seed(42) set globally in a security context. LLM Detection: **High.** An LLM can identify usage of the random module for things like tokens or keys and knows this is discouraged (the Python docs explicitly warn that random is not for crypto ([random — Generate pseudo-random numbers — Python 3.13.3 ...](https://docs.python.org/3/library/random.html" \l ":~:text=random%20%E2%80%94%20Generate%20pseudo,Matsumoto))). It will suggest using the secrets or os.urandom module instead. Patterns like a fixed seed (seed(…)) are also easily noticed and can be flagged as making the randomness predictable.
* **Ignoring Certificate Validation**: Disabling SSL/TLS certificate verification in HTTP clients or other network calls. This is a misuse because it allows man-in-the-middle attacks. For example, using requests.get(url, verify=False) will not verify the server’s TLS certificate, exposing the connection to impersonation ([Cryptographic Failures: Understanding and Preventing Vulnerabilities | by Ajay Monga | Medium](https://medium.com/@ajay.monga73/cryptographic-failures-understanding-and-preventing-vulnerabilities-91c8b2c56854" \l ":~:text=Missing%20or%20Improper%20Validation)). Example: Code that connects to an API with urllib3.disable\_warnings() and verify=False, or an ssl.\_create\_unverified\_context() for an HTTPS connection. LLM Detection: **High.** The presence of flags or calls explicitly turning off verification is a clear signal. The model can pick up verify=False or functions named like “unverified\_context” and know that this is a security issue (it effectively defeats the purpose of SSL). It will flag those lines as vulnerabilities, often suggesting to remove that flag or properly validate certificates.
* **Poor Key Management Practices**: This covers using weak keys, reusing keys in multiple places, or not rotating keys. For instance, using a short encryption key like a simple password, or using the same key for all customers and never changing it. Example: Deriving an encryption key from user input directly (making it guessable), or embedding a constant key that the code comments indicate is for “demo only” but ends up in production. LLM Detection: **Moderate.** Some of these issues are harder to detect without context (like recognizing a key is too short might require the LLM to infer bit-length). However, obvious signs such as a constant key value or reuse of one key in multiple functions could be caught. The LLM may not always know if a key is reused across contexts unless it sees it, but it will warn about any key that doesn’t seem securely generated or stored.

## Code Quality & Dangerous Patterns

* **Use of** eval()**/**exec() **on Untrusted Input**: Dynamically executing code from strings or input is a well-known dangerous practice. Python’s eval() will execute any code given to it ([Command injection in Python: examples and prevention | Snyk](https://snyk.io/blog/command-injection-python-prevention-examples/" \l ":~:text=Using%20,os)), so if an attacker can influence that input, they can run arbitrary Python commands (even OS commands via modules). Example: data = input("Expression: "); eval(data). An attacker entering \_\_import\_\_('os').system('rm -rf /') would execute that system command ([Command injection in Python: examples and prevention | Snyk](https://snyk.io/blog/command-injection-python-prevention-examples/" \l ":~:text=Using%20,os)). LLM Detection: **High.** The LLM readily recognizes eval( or exec( usage. These functions are almost always risky; the model will flag them especially if the evaluated string comes from an input or variable. It understands from context (and common security knowledge) that using eval without strict controls is akin to a remote code execution vulnerability, and it will point that out.
* **Insecure Deserialization (Pickle)**: Using Python’s pickle module (or similar serialization libraries) to load data from untrusted sources. pickle.loads() can instantiate arbitrary objects and even execute functions during unpickling, which can lead to remote code execution if the data is crafted by an attacker ([Unsafe Deserialization in Python | SecureFlag Security Knowledge Base](https://knowledge-base.secureflag.com/vulnerabilities/unsafe_deserialization/unsafe_deserialization_python.html" \l ":~:text=return%20%27OK%27)) ([Unsafe Deserialization in Python | SecureFlag Security Knowledge Base](https://knowledge-base.secureflag.com/vulnerabilities/unsafe_deserialization/unsafe_deserialization_python.html" \l ":~:text=class%20Pickle,%2Ftmp%2Fproof)). Example: A Flask endpoint that receives a file upload and does user\_obj = pickle.loads(file\_bytes) directly. An attacker can include a payload in the pickle that runs OS commands on deserialization ([Unsafe Deserialization in Python | SecureFlag Security Knowledge Base](https://knowledge-base.secureflag.com/vulnerabilities/unsafe_deserialization/unsafe_deserialization_python.html" \l ":~:text=A%20malicious%20user%20could%20craft,pickle.loads)). LLM Detection: **High.** The model will flag usage of pickle.load/loads on data that could come from outside (files, network, etc.). It knows from security literature that untrusted pickle = RCE. The patterns in code (importing pickle and then loading data) are straightforward to catch. It would likely recommend using safer serialization (like JSON) or ensure the source is trusted ([Unsafe Deserialization in Python | SecureFlag Security Knowledge Base](https://knowledge-base.secureflag.com/vulnerabilities/unsafe_deserialization/unsafe_deserialization_python.html" \l ":~:text=Prevention)).
* **Unsafe YAML or XML Loading**: Similar to pickle, using libraries like PyYAML’s yaml.load with a full loader can execute arbitrary object constructors ([Be Careful When Using YAML in Python! There May Be Security Vulnerabilities - DEV Community](https://dev.to/fkkarakurt/be-careful-when-using-yaml-in-python-there-may-be-security-vulnerabilities-3cdb" \l ":~:text=The%20YAML%20,attack%20surface%20for%20malicious%20payloads)). For instance, yaml.load(untrusted\_data) without SafeLoader can be abused to run code (as YAML supports defining Python objects). XML libraries can also be abused (XXE attacks) if external entity resolution is enabled. Example: yaml.load(user\_yaml\_string) in a web API – an attacker can embed !!python/object/apply:os.system ["ls"] in the YAML to execute commands ([Be Careful When Using YAML in Python! There May Be Security Vulnerabilities - DEV Community](https://dev.to/fkkarakurt/be-careful-when-using-yaml-in-python-there-may-be-security-vulnerabilities-3cdb" \l ":~:text=The%20YAML%20,attack%20surface%20for%20malicious%20payloads)) ([Be Careful When Using YAML in Python! There May Be Security Vulnerabilities - DEV Community](https://dev.to/fkkarakurt/be-careful-when-using-yaml-in-python-there-may-be-security-vulnerabilities-3cdb" \l ":~:text=import%20yaml%20from%20yaml%20import,Loader%2C%20UnsafeLoader)). LLM Detection: **High.** The LLM can identify the use of yaml.load (especially if no Loader= parameter specifying safe mode) and flag it as dangerous. This is a known pitfall (the model likely has seen discussions of CVE-2020-14343 for PyYAML). Similarly, if it sees code enabling XML entities or using outdated XML parsers without protections, it will issue a warning. These patterns (like calling XMLParser(resolve\_entities=True)) are specific and catchable.
* **Use of** assert **for Security Logic**: Relying on Python assert statements to enforce important conditions (like checking user roles or inputs) is risky. In optimized mode (python -O), asserts are stripped out entirely, meaning any “check” can be bypassed ([The dangers of assert in Python | Snyk](https://snyk.io/blog/the-dangers-of-assert-in-python/" \l ":~:text=Assert%20helps%20developers%20debug%20Python,applications%20if%20not%20used%20safely)). Attackers can simply exploit the fact that the assert might not execute in production, leading to hidden backdoors or disabled security checks ([The dangers of assert in Python | Snyk](https://snyk.io/blog/the-dangers-of-assert-in-python/" \l ":~:text=applications%20if%20not%20used%20safely)). Example: assert user.is\_admin, "Only admins allowed" before a critical operation – if the application is run with optimizations, this check does nothing, and even non-admins can proceed. LLM Detection: **Moderate.** The LLM will catch an assert being used where a proper if/raise or exception handling should be. It understands that asserts are removed in optimized bytecode ([The dangers of assert in Python | Snyk](https://snyk.io/blog/the-dangers-of-assert-in-python/" \l ":~:text=Assert%20helps%20developers%20debug%20Python,applications%20if%20not%20used%20safely)). It will likely flag any assert in a web app context that isn’t strictly for debugging. However, it needs to infer whether the assert is guarding something security-relevant (which in many cases is clear from the message or context). It may warn generically about use of assert for anything other than tests or debugging.
* **Other Dangerous Patterns**: This includes things like overly broad exception handling (catching all exceptions and continuing, which can mask security issues or lead to unintended program state), or using functions like input() in Python 2 (which acted like eval). It also includes spawning subprocesses with shell=True (already covered under injection) or writing temporary files insecurely. Example: try: ... except Exception: pass – this would swallow errors, possibly hiding an authentication bypass or causing the program to continue in an invalid state. LLM Detection: **Moderate.** The model can notice a bare except: or except Exception with no handling and point out that it’s a bad practice. While not a vulnerability by itself, it’s a quality issue that can facilitate vulnerabilities. The LLM’s code analysis might recommend specifying exception types or at least logging the error. For temp files, if it sees use of tempfile.mktemp() (known to be insecure), it can flag that too. These are more code hygiene issues, but an AI-based scanner can include them as they often correlate with sloppy security practices.

## Error Handling & Logging Risks

* **Logging Sensitive Information**: Recording confidential data in logs (passwords, personal info, keys) can lead to leaks. Logs are often plain text and accessible to developers or in worst cases to attackers who get file read access ([10 Best Practices for Logging in Python | Better Stack Community](https://betterstack.com/community/guides/logging/python/python-logging-best-practices/" \l ":~:text=Sensitive%20data%20should%20be%20kept,and%20can%20contain%20sensitive%20information)). For example, printing a user’s password or credit card number in an error log means anyone with log access (or an intruder who finds the logs) can steal that data ([10 Best Practices for Logging in Python | Better Stack Community](https://betterstack.com/community/guides/logging/python/python-logging-best-practices/" \l ":~:text=Sensitive%20data%20should%20be%20kept,and%20can%20contain%20sensitive%20information)) ([10 Best Practices for Logging in Python | Better Stack Community](https://betterstack.com/community/guides/logging/python/python-logging-best-practices/" \l ":~:text=,that%20your%20logging%20system%20is)). Example: logger.error(f"Login failed for user {username} with password {password}") – this line writes the actual password to a log file. LLM Detection: **High.** An LLM can scan for log statements (logging.info/error/debug or print statements) that include sensitive keywords or variables. If it sees something like password or secret being concatenated or formatted into a log message, it will flag it. The AI understands the principle of data minimization in logging ([10 Best Practices for Logging in Python | Better Stack Community](https://betterstack.com/community/guides/logging/python/python-logging-best-practices/" \l ":~:text=risk%20of%20sensitive%20data%20being,exposed)) and will warn about any instance of sensitive data being logged. It may also flag large dumps of objects (which might include secrets by accident) and suggest scrubbing or masking.
* **Verbose Error Messages / Debug Mode**: Exposing too much information in error responses or running in a debug configuration in production. For instance, Flask’s debug=True enables the interactive debugger and shows stack traces with local variables on any error – a **major risk** if an attacker triggers an error ( [flask — code injection | Precaution](https://docs.securesauce.dev/rules/PY507" \l ":~:text=considered%20bad%20for%20several%20reasons%3A)). Detailed exceptions can reveal file paths, code snippets, or config values. Example: A Flask app left in debug mode will display a full traceback and even allow code execution via the Werkzeug console to anyone who can access the web interface ( [flask — code injection | Precaution](https://docs.securesauce.dev/rules/PY507" \l ":~:text=1,unpredictable%20in%20a%20production%20environment)). Similarly, returning e.\_\_repr\_\_() in an API error response might show internal secrets. LLM Detection: **High.** The model recognizes the telltale signs of debug mode (for Flask, seeing app.run(debug=True) ( [flask — code injection | Precaution](https://docs.securesauce.dev/rules/PY507" \l ":~:text=from%20flask%20import%20Flask)), or Django DEBUG setting, etc.). It will flat-out label this as a vulnerability because it’s widely known that debug mode should never be enabled in production ( [flask — code injection | Precaution](https://docs.securesauce.dev/rules/PY507" \l ":~:text=Using%20the%20,considered%20bad%20for%20several%20reasons)). For overly verbose errors, if it sees code catching exceptions and printing or returning the exception message/stack, it will warn about information leakage. The AI can connect that stack traces or debug info could help attackers by revealing environment details.
* **Failure to Handle Exceptions (Stability Issues)**: Not catching exceptions at all in critical areas can cause crashes, leading to denial of service. While this is more of a reliability issue, it becomes a security concern if an attacker can intentionally cause an unhandled exception to crash the app. Example: A function that assumes a file exists and does open(filename) without try/except – an attacker deleting the file or using an unexpected filename could crash the process. LLM Detection: **Low.** It’s hard for an LLM to proactively identify where an exception should be caught if the code doesn’t already attempt to do so. However, it might identify functions that call risky operations (I/O, network, etc.) and do not handle any exception, and point out that a crash could result. This is usually a secondary concern in AI code scanning, so it may not always flag it unless obvious.
* **Logging or Printing Too Much Detail**: Similar to above, but even things like logging full stack traces or objects on error can leak data (for example, printing an object might call its \_\_repr\_\_ which dumps sensitive fields). Example: Catching an exception and doing logger.exception(e) (which prints a full traceback) in a context where the log might be exposed. LLM Detection: **Moderate.** The LLM knows that logger.exception will include a traceback. It might not flag this as an outright vulnerability (since in many cases it’s fine if logs are secured), but in a context where logs could be accessed by users, it might note it. Generally, the AI will focus on clear sensitive data exposure rather than generic logging of errors.

## Dependency & Package Risks

* **Use of Outdated/Vulnerable Libraries**: Depending on third-party packages with known security vulnerabilities or that are well past their update lifecycle. In a Python project, this could mean an old Django/Flask version missing a critical patch or an insecure library like a version of requests with a known bug. These components are “vulnerable and outdated” and put the application at risk ([OWASP Top 10 — A06:2021 Vulnerable and Outdated Components | by Madhumathi chamarthi | Medium](https://medium.com/@madhuhack01/owasp-top-10-a06-2021-vulnerable-and-outdated-components-2dfe74e60cea" \l ":~:text=What%20Are%20Vulnerable%20and%20Outdated,in%20use%20within%20an%20application)). Example: The code imports django==1.6 in requirements (which is years old and has known CVEs), or uses a package that hasn’t been updated in a long time (no support). LLM Detection: **Moderate.** If the LLM is provided with the dependency list (requirements.txt or Pipfile with versions), it can cross-reference them against its training knowledge of known issues or simply realize the version is very old. It might say “Library X v1.2.3 is outdated; newer versions address security flaws”. However, without explicit version info in code, the model might not know the app is using a vulnerable version. Enabling an LLM with a vulnerability database would improve detection here. In prompt-engineered analysis, the AI can still flag the import of libraries that are commonly dangerous or deprecated (e.g. using Crypto library instead of cryptography, or an abandoned project).
* **Insecure Configurations of Dependencies**: Using libraries in an unsafe way by disabling their security features or using dangerous options. One example is decoding JSON Web Tokens (JWT) without verifying the signature – e.g., using a JWT library’s verify=False or not providing a secret key, which would allow any token to be accepted as valid. Another example is using an ORM with automagic migrations that drop access controls. Example: jwt.decode(token, options={"verify\_signature": False}) – this would trust tokens blindly, enabling attackers to forge JWTs and impersonate users. LLM Detection: **High.** The AI can detect when code calls security-related APIs with parameters that disable security. It knows that in PyJWT or similar libraries, skipping verification is a critical flaw. It will flag code patterns like a JWT decode function where no secret or public key is provided, or a parameter explicitly set to false for verification. Similarly, any use of a library’s “debug” or “insecure” modes (like SQLAlchemy echoing passwords, or param verify\_ssl=False in an LDAP bind) can be recognized by the model. These are usually clearly indicated in code, making detection straightforward.
* **Unsafe or Malicious Packages**: Importing modules that are known to be potentially unsafe or using wildcard imports that might shadow built-in secure functions. For instance, installing a package that is a typo-squatted name (e.g. importing reqeusts instead of requests) could introduce malicious code. While this is more of a supply-chain issue, a code scanner might notice unusual package names or usage patterns. Example: import reqeusts (typo intended) – an LLM could realize this is not the standard library and flag the possibility of a malicious dependency. LLM Detection: **Moderate.** The model’s ability to catch this depends on its knowledge of common libraries. If it sees an import that looks very similar to a popular one but not quite, it can suspect a typo and the presence of a rogue package. It can also flag if a package is doing something odd on import (though that requires analyzing package code, which might be out of scope). Overall, the LLM can warn about non-standard or surprising imports, but confirming malicious intent might need integration with a vulnerability database or sandbox analysis.
* **Lack of Dependency Pinning and Integrity**: Not exactly a vulnerability in code, but a practice risk – if the project doesn’t pin versions (allowing package>=1.0 without an upper bound) or doesn’t verify package signatures/hashes, an attacker could slip a malicious update or a compromised package version. Example: A requirements file with flask>=2.0 could one day pull in a 2.x version that has a zero-day. LLM Detection: **Low.** The model might generically recommend pinning versions or using hashes (like Pip’s requirements with --hash), but it won’t treat this as an outright vulnerability it can see – it’s more of a process weakness. In an advisory role, an LLM could suggest improving this when asked about dependency management.

## Performance & Maintainability Issues

* **N+1 Database Query Pattern**: A performance anti-pattern where the code queries the database in a loop (N times), instead of doing a set-based query, leading to N+1 queries total ([N+1 Queries in Django: How They Affect Performance and How to Fix and Find Them | by Richard Luo | Level Up Coding](https://levelup.gitconnected.com/n-1-queries-in-django-how-they-affect-performance-and-how-to-fix-and-find-them-8123f1500aae" \l ":~:text=The%20N%2B1%20query%20problem%20can,number%20of%20database%20queries%2C%20which)). This can severely slow down an application for large N. Example:

users = User.query.all()

for u in users:

orders = Order.query.filter\_by(user\_id=u.id).all()

...

Here, if there are N users, it executes 1 query to get users + N queries (one per user) for orders. LLM Detection: **High.** The LLM can analyze the code flow and spot that a query is executed inside a loop. It knows this pattern is likely unintended and can suggest using a join or eager loading (for instance, using .select\_related() or a single query with a join) ([N+1 Queries in Django: How They Affect Performance and How to Fix and Find Them | by Richard Luo | Level Up Coding](https://levelup.gitconnected.com/n-1-queries-in-django-how-they-affect-performance-and-how-to-fix-and-find-them-8123f1500aae" \l ":~:text=Fortunately%2C%20Django%20provides%20a%20solution,data%20in%20the%20original%20query)). Tools and research have shown that AIs can detect N+1 issues by recognizing query-in-loop constructs. The prompt would flag this as a performance vulnerability because it’s something an AI-based reviewer is well-suited to catch from static code.

* **Redundant or Duplicate Logic**: Repeated code blocks or logic scattered in multiple places, making maintenance error-prone. While not a security flaw per se, redundant logic can cause inconsistencies (one part might be patched for a security fix, while the duplicate elsewhere is forgotten). Example: Input validation logic copied into three different endpoints – if a rule changes, one endpoint might not update, creating a security gap. Or simply the same complex calculation copy-pasted in several functions (violating DRY principle). LLM Detection: **Moderate.** The AI can use its understanding of the code structure to identify very similar or identical code snippets. It might say “these two functions contain almost the same code, consider refactoring.” For prompt-based analysis, if code is provided file by file, it might not cross-compare well; but a whole-repository analysis mode could enable it to flag duplicate code. It will reliably catch exact duplicates or very close similarities. This improves maintainability and indirectly security (by reducing the chance of one instance not getting a fix).
* **Inefficient Algorithms or Operations**: Code with obvious inefficiencies like unnecessarily nested loops, expensive computations on each request, or not using optimized library functions. For example, assembling strings with += in a loop (which in Python causes lots of intermediate objects) instead of using .join(), or sorting data multiple times. Example: A function that recalculates something heavy inside a loop, or opens and reads a file repeatedly for each record instead of once. LLM Detection: **Moderate.** The LLM’s understanding of Big-O complexity and Python performance idioms lets it notice some inefficiencies. It may flag nested loops over large datasets, or point out when a standard library method could do the job faster. In terms of prompt-engineered scanning, it will likely point out the clearest issues (like repeated string concatenation in a loop, which is a known bad practice). It might not fully optimize your code, but it can highlight spots that are unusually costly relative to what they accomplish.
* **Scalability and Maintainability Concerns**: Broadly, patterns that make code hard to scale or maintain – such as giant monolithic functions, lack of modularization, or no comments/documentation on complex logic. While these aren’t “vulnerabilities,” they are issues that an AI code assistant can detect. Example: A single 500-line view function handling all sorts of logic (violating separation of concerns), or using global variables that introduce side effects across modules. LLM Detection: **Moderate.** The AI can measure function length, number of responsibilities, and usage of globals. It will often recommend splitting large functions or avoiding global state. These suggestions improve code quality and reduce the likelihood of security bugs slipping in unnoticed. In prompt mode, the AI could output a warning like: “This function is very long or does many things – consider refactoring for clarity and maintainability.” Such issues are not cut-and-dry, so the AI’s advice may be more subjective here, but still valuable for long-term health of the codebase.

## Opportunities for Future Expansion

* **System Configuration & Infrastructure Scanning**: Extend checks beyond application code to system-level configs. For example, scanning for dangerous settings in **server configurations** (Apache/Nginx configs, database configs) or environment settings. This could catch issues like directory listings enabled, debug modes on in production environments, or improper file permissions on config files. An AI could parse config files or Docker Compose files to identify these misconfigurations.
* **Container and Deployment Security**: Analyze Dockerfiles and Kubernetes manifests for security risks. This includes detecting if Docker images run as root, if unnecessary ports are exposed, missing security hardening (no CMD to a non-root user, using outdated base images), or if Kubernetes pod specs disable security features (like running privileged containers). Future iterations could have prompt templates for Dockerfile best practices (e.g., no plaintext secrets, proper use of HEALTHCHECK, minimal base images) and K8s YAML checks (like ensuring resource limits, avoiding latest tags, and not mounting excessive host volumes).
* **Cloud/IAM Policy Analysis**: Incorporate cloud infrastructure scanning such as AWS IAM policies, GCP IAM roles, Azure RBAC, etc. This would involve analyzing JSON/YAML policy documents to spot things like wildcard permissions ("\*" actions or resources), overly permissive roles, or missing multi-factor authentication enforcement. An AI can be tuned to understand the principle of least privilege and flag IAM policies that violate it. Additionally, checking Terraform or CloudFormation files for cloud resource misconfigurations (open S3 buckets, public-facing databases, etc.) would be a natural expansion.
* **Supply Chain & Build Pipeline Security**: Future prompts could target the software supply chain. This means scanning CI/CD pipeline definitions (GitHub Actions, Jenkinsfiles) for insecure practices such as storing secrets in plain text, not verifying build artifacts, or downloading dependencies from unofficial sources. Also, detecting use of vulnerable package versions (via an integrated vulnerability database) or verifying signatures of dependencies can prevent supply-chain attacks. An AI could, for example, read a GitHub Actions workflow and warn if dependencies are installed without checksum verification, or if deployment keys are injected improperly.
* **Dynamic and Runtime Analysis**: Augment static code prompts with runtime behavior analysis prompts. For instance, analyzing application logs or monitoring data for suspicious patterns (like frequent login errors indicating a brute force attack, or memory spikes indicating a leak). While this veers into dynamic analysis, an LLM could assist by correlating code with runtime outputs. In the future, prompts might be designed to reason about execution traces or test results – effectively combining static analysis with insights gleaned from running the code (fuzz testing outcomes, performance profiling, etc.).
* **Cross-Language and Full-Stack Security**: Expand the scanning to front-end code (JavaScript), mobile app code, or other services that interact with the Python backend. Security issues often span multiple layers – e.g., an insecure backend API combined with a vulnerable front-end could be exploited in tandem. A comprehensive future tool could use LLMs to analyze how data flows from the frontend (perhaps checking if input validation is consistently applied both client and server side) or to ensure that security headers set in the backend (like CSP, HSTS) align with front-end expectations. Additionally, checking mobile app code for hardcoded API keys that correspond to the backend would close the loop on end-to-end scanning.

By broadening the scope to these areas, an AI-based analysis system would provide a more holistic security assessment, catching not only code issues but also configuration and deployment weaknesses. Each of these expansions would come with its own set of prompt templates and pattern checks, many of which align with known best practices and security benchmarks (like CIS benchmarks for Docker/K8s, or OWASP Secure Configuration guides). The modular nature of LLM prompts means these capabilities could be added incrementally, ultimately giving developers a **one-stop, AI-powered security review** covering code **and** infrastructure.