DevForth

How to Build Low-Latency, Cost-Efficient Agentic Applications

A practical founder-focused guide to the product and architecture decisions that make AI agents faster, cheaper, more reliable, and easier to scale.

November 2026

Contents

Cheap demos often turn into expensive products . . . 3
Context and cost control . . . 3
- Do not pay for the same prompt again and again . . . 3
- Do not put the whole product into one giant prompt . . . 4
- Track tokens like product costs, not just technical numbers . . . 4

Tools and execution safety . . . 5
- Give the agent clear actions, not a wall of documentation . . . 5
- Put dangerous actions behind confirmation . . . 6

Data, memory, and retrieval . . . 7
- Do not feed the agent noisy backend data . . . 7
- Long chat history is not memory . . . 8
- Good memory is organized by purpose . . . 9
- Search quality matters more than many teams expect . . . 9
- Bad document splitting can make the agent miss the answer . . . 10
- The agent is only as good as the knowledge it receives . . . 11
- Teach the agent when not to answer . . . 11

Product experience and scaling . . . 12
- Do not leave language handling to chance . . . 12
- Do not lock the product into one model forever . . . 13
- More agents do not automatically mean a better product . . . 13

What this means in practice . . . 14

Cheap demos often turn into expensive products

An AI agent can look great in a demo. It answers quickly, sounds smart, follows instructions, and gives the feeling that the product is almost ready. But a five-minute demo and a real product are very different things.

The real question is not whether the agent can impress someone once. The real question is whether it can stay fast, reliable, and economically reasonable when real users start using it every day.

This is where architecture becomes a business question. Architecture affects how much each request costs, how fast the product feels, how often the agent makes mistakes, and how hard it will be to improve the product later.

Bad architecture does not always break the product immediately. More often, it quietly makes the product slower, more expensive, and harder to maintain. That is why agent architecture should not be treated as something to fix "later". If the product grows, every inefficient decision grows with it.

Context and cost control

Do not pay for the same prompt again and again

Most agent products send a lot of repeated text to the model in every request. This may include system instructions, product rules, safety policies, descriptions of available actions, formatting rules, and examples. At first, this does not feel like a problem. The product works, the agent responds, and the team keeps adding more instructions whenever something goes wrong.

But every repeated block has a cost. If the same large instruction is sent again and again without being cached, the business keeps paying for the same text as if it were new. This is not just a technical detail. It directly affects the cost of every user action.

A good agent product should be designed so that stable, repeated parts of the prompt remain stable. Permanent instructions should not change randomly between requests, and large repeated blocks should not be regenerated differently every time without a clear reason.

The exact caching rules depend on the AI provider. In many cases, caching works best when the repeated part is large enough and appears in a stable position. Cached tokens usually still appear in usage reports, but they may be cheaper or faster to process.

This means the team should not look only at the total number of input tokens. It should also understand which tokens are cached and which are not. If a product has a large prompt but almost no cached input, it often means the prompt is not organized well.

The result is predictable: higher cost, slower responses, and weaker margins.

Do not put the whole product into one giant prompt

At an early stage, it is normal to keep things simple. You can give the agent one large instruction, describe how it should behave, connect a few actions, and quickly get a working prototype.

An action, often called a tool, is something the agent can do outside the chat: read data from a database, create a record, send an email, fetch a list of orders, or change settings. For a small product, one general prompt and a few actions may be enough.

The problem starts when the product grows. More features appear. More actions are added. More rules are needed. More edge cases show up. If everything keeps going into one large prompt, the system becomes expensive, noisy, and difficult to control.

A better approach is to load only the instructions and actions that are relevant to the current task. Some teams call these separate instruction modules "skills". For example, there can be a billing skill, analytics skill, support skill, or dashboard skill. Each one contains only the rules, examples, and actions needed for that specific area.

The important idea is simple: the agent should not see the entire product manual every time it answers one user request. It should receive only what it needs right now.

This is not always a built-in feature of an AI provider. In many custom products, the team has to implement it itself: by routing tasks, loading the right instruction module, searching for the right tools, or using a separate orchestration layer.

When this is done well, the agent receives less irrelevant context. The product uses fewer tokens, saves space in the model context, and reduces the chance that the agent will choose the wrong action.

A product should not become more expensive and less stable every time the team adds a new feature.

Track tokens like product costs, not just technical numbers

It is not enough to know that one request costs a certain amount. The team should understand where that cost comes from.

A useful cost breakdown answers practical questions:

- How much text is sent as the main instruction?
- How much comes from chat history?
- How much comes from documents?
- How much comes from tool results?
- How much input is cached?
- How much is generated as the final answer?

Without this visibility, cost optimization becomes guesswork.

There is an important detail here: the AI provider usually does not give a full breakdown by every part of your prompt. It may show total input and output tokens, cached tokens, or other usage details, but it usually will not automatically tell you how much came from “instructions”, “tools”, “documents”, or “history”.

The product team often has to measure this at the application level. That means tagging parts of the prompt, logging the size of retrieved context, counting tool result size, and tracking which blocks have the biggest impact on cost and response time.

These metrics are useful not only for saving money. They also make debugging much easier.

If the agent takes too long, makes unnecessary tool calls, or goes in circles, the team needs to see where the problem is. Is it the main instruction? The action description? The memory? The retrieved document? The task itself?

Without metrics, the team is guessing. With metrics, the team can improve the system intentionally.

Tools and execution safety

Give the agent clear actions, not a wall of documentation

If the agent can perform actions, those actions need clear descriptions. The agent should understand what the action does, when it should be used, what information it needs, what can go wrong, and what it will return.

The product also needs clear contracts for these actions: required fields, allowed values, expected formats, and important limitations. But the description of an action should not become a full product manual.

Too much text creates noise. The model may focus on irrelevant details, miss the important constraint, or call the action in the wrong situation. At the same time, “short” does not mean “empty”.

A bad action description can be short and still useless. For example, “creates widget” does not tell the agent when to use it, what widget types exist, what information is required, or when this action should not be used.

A good description is compact but useful. It gives the agent enough context to use the action correctly without forcing it to read a long manual every time.

If an action is simple, a clear description may be enough. If an action is complex, examples are often necessary. For example, if the agent has to create a dashboard widget, choose data, configure a chart, and place it into a layout, a list of fields may not be enough. The agent can see the fields, but

still misunderstand the correct pattern.

This is where a few good examples help. The balance matters: no examples means more mistakes, but too many examples turn the instruction module into another giant prompt.

A practical rule is simple: action descriptions should be compact and precise, while examples should be added only where they actually reduce errors.

Another common mistake is giving the agent several tools that look similar or can perform the same task in different ways.

For a developer, the difference between these tools may be clear. For the agent, it may create unnecessary ambiguity. If one action can create a record, another can update a record, and a third can "save configuration" in a broader way, the agent may choose the most general action even when a more specific one would be safer.

This is especially risky when broad tools can perform many types of changes. A general tool may look convenient because it reduces the number of APIs, but it gives the agent more room to make a wrong decision.

In many cases, smaller and more atomic tools are easier for the agent to use correctly. A tool that renames a dashboard group, a tool that adds a widget, and a tool that updates a widget's chart configuration are easier to reason about than one large tool that can change everything.

This does not mean every tiny operation needs a separate tool. Too many tools can also create noise. The point is that each action should have a clear purpose, clear boundaries, and a low chance of being confused with another action.

A good tool set should make the correct action obvious.

Put dangerous actions behind confirmation

An agent can do many useful things, but not all actions have the same level of risk.

Some actions are mostly safe. For example, reading data, searching through documents, summarizing information, or explaining a report usually does not change the state of the product. If the agent makes a mistake, the answer may be wrong, but the underlying data remains unchanged.

Other actions are different. Editing records, deleting data, changing user permissions, sending emails, cancelling orders, modifying billing settings, or publishing content can have real consequences. The risk depends on the value of the data and the business process around it, but the general rule is simple: actions that change important state should not be treated like normal chat responses.

A common first solution is to handle this in the prompt. The team tells the agent: before making a sensitive change, explain what you are going to do and ask the user for confirmation.

This is better than nothing, but it is not strong enough for production. A prompt rule can work well most of the time and still fail in the one case where failure is expensive. The agent may misunderstand the situation, classify the action as safe, skip the confirmation, or follow a different instruction from the surrounding context.

A safer design is to put confirmation into the product itself, not only into the prompt.

For example, a dangerous action can be implemented so that the first call only prepares a preview: what will be changed, which records are affected, and what the result will be. The actual change happens only after the user confirms it. In this design, the agent does not need to be perfectly reliable about safety. Even if it tries to perform the action immediately, the tool or backend can stop the operation and require confirmation first.

This is especially important for irreversible or hard-to-reverse actions. Deleting customer data, changing financial settings, sending external messages, or modifying production configuration should not depend only on the agent remembering one sentence from the prompt.

The practical pattern is simple: let the agent help with preparation, analysis, and explanation, but put high-risk execution behind a confirmation step controlled by the application.

Human-in-the-loop does not mean the user must approve every small step. If every action requires confirmation, the product becomes annoying and slow. The goal is to separate safe actions from risky ones and require confirmation only where a mistake can create real damage.

A good agent product should make dangerous actions possible, but not casual.

Data, memory, and retrieval

Do not feed the agent noisy backend data

Most applications already have useful data in formats that are convenient for the backend. This can be database records, API responses, logs, configuration objects, user profiles, or tool results. Very often this data exists as JSON.

JSON is a good format for machines. It is strict, easy to validate, easy to parse, and useful when the system needs an exact structure. But raw JSON is not always the best format for the agent to read.

If the agent needs to understand the data, compare fields, explain something to the user, or make a decision, the data should often be converted into a more readable form first.

The practical rule is simple: do not pass data to the agent in the format that is easiest for the backend. Pass it in the format that is easiest to understand.

For example, instead of sending a large nested JSON object with internal IDs, null values, timestamps,

debug fields, and irrelevant metadata, the system can show the important parts in a compact readable format:

```
Customer:
  Name: John Smith
  Plan: Pro
  Status: Active
  Renewal date: 2026-07-15

Recent activity:
  - Created 3 projects
  - Invited 2 teammates
  - Opened 1 support ticket

Relevant warning:
  Payment method expires next month
```

This is easier to scan, easier to debug, and usually easier for the agent to use.

The same applies to results returned by tools. A tool can return strict JSON internally, but before that result is sent back into the model context, the application can render only the important parts in a cleaner format.

This does not mean JSON should never be shown to the model. If the agent must edit exact JSON, generate a valid JSON response, call a function, or preserve a strict structure, JSON is often the right format.

But for context, summaries, retrieved information, explanations, and product knowledge, readable text is often a better default.

A good agent system separates internal data format from model context format. Internally, the back-end can use JSON, database rows, typed objects, or API responses. Externally, the model should receive a clear, compact, task-focused version of the data.

The goal is not to make the context look pretty. The goal is to reduce noise, preserve meaning, and make the agent less likely to miss the important part.

Long chat history is not memory

Many teams simply send the entire conversation history back to the model. This feels safe: the model sees everything that happened before, so it seems like nothing can be lost.

But in production, this quickly becomes expensive noise. Full history increases cost, slows down responses, and makes the model less focused.

A better pattern is simple: keep the last few messages in full, and compress older history into a short summary. The agent usually does not need every sentence from the previous conversation. It needs the current task, recent context, important decisions, and relevant user preferences.

Memory should help the agent. It should not be an archive of every message that ever appeared in the conversation.

Otherwise, the product pays not for useful context, but for a pile of old text.

Good memory is organized by purpose

Memory should not be one large storage bucket for everything. Different types of information should be stored and used differently.

For example, the agent may need short-term context for the current task. It may need a summary of what happened in the current session. It may need stable product knowledge. It may also need rules about how certain workflows should be handled.

Teams may name these memory layers differently. The names are not the important part. The important part is separation.

If all memory is mixed together, the agent starts pulling old or irrelevant information. Prompts become larger, mistakes become harder to understand, and debugging becomes more painful.

A strong memory system does not send everything to the model at once. It retrieves only what is useful for the current task.

Memory is not just storage. Memory is the ability to find the right information at the right moment.

Search quality matters more than many teams expect

Many agent products need to search through documents, help centers, policies, user data, or internal knowledge bases. This is often called retrieval or RAG.

The idea is simple: when the agent needs knowledge, the system first searches for relevant information and then gives that information to the model as context.

When an agent gives a poor answer, the team often assumes the model failed. But very often the real problem is simpler: the system did not find the right context.

Good search should usually combine two things. First, it should search by meaning. This helps when the user asks a question in different words than the document uses. Second, it should search for exact matches. This matters for names, IDs, emails, codes, legal terms, product names, pricing plans, and other precise values.

If the system relies only on meaning-based search, it may find something that sounds similar but miss the exact document or exact record. If it relies only on keyword search, it may miss useful information written in different words.

Strong retrieval usually combines both.

The business impact is simple: if the system cannot find the right context, the agent cannot reliably give the right answer.

Bad document splitting can make the agent miss the answer

Documents are usually too large to send to the model all at once. So systems split them into smaller parts, often called chunks, and search through those parts.

This sounds simple, but it can easily go wrong. The full answer may be split across several chunks. The system finds one relevant part, but that part contains only half of the answer. The beginning or the end remains in a neighboring chunk.

For the user, this looks as if the agent is answering superficially or does not know the full information.

The solution is not just to connect a vector database. The system needs to split documents carefully, keep enough surrounding context, and respect the structure of the document: headings, sections, tables, lists, and step-by-step instructions.

A practical rule is to keep chunks in a format that is readable not only for the machine, but also for humans. Markdown is often a good default for documentation, policies, support articles, technical guides, and product knowledge. It is simple, easy to review, easy to diff in version control, and clear enough for both humans and LLMs.

This does not mean Markdown is always the best format for everything. Strict structured data may be better represented as JSON or CSV. Complex visual documents may need special extraction. Large tables should be handled carefully so the relationship between columns, rows, and headers is not lost.

For example, it is risky to split a multi-step instruction in a way where the agent sees only steps 3–5 without the conditions described in steps 1–2.

Similarly, a table or pricing policy should not be split like plain text if that breaks the relationship between columns, conditions, and values. If a table must be split, the chunk should preserve the table header or enough surrounding context for the agent to understand what the values mean.

RAG quality depends not only on the model. It also depends on how the system searches, splits, formats, and passes context.

The agent is only as good as the knowledge it receives

If an agent answers based on documents, policies, or an internal knowledge base, the quality of its answers depends not only on the model. It also depends on the quality of the knowledge base.

If the knowledge base contains old instructions, duplicates, conflicting rules, or drafts, the agent may retrieve exactly those and answer incorrectly. For the user, this will look like an AI mistake. But the real cause is often simpler: the system gave the agent bad context.

This cannot be solved with a prompt alone. The knowledge base needs structure.

Current documents should be marked as current. Old ones should be archived. Drafts should be separated from approved materials. Documents should have versions, dates, owners, and clear status.

The search system should also understand these differences. The agent should not trust an old PDF, a random uploaded file, and an approved policy equally. The system should prefer sources that are current, approved, relevant, and reliable.

The practical pattern is simple: first organize the knowledge base, then teach the system to retrieve not just similar text, but the most current and trustworthy source.

Otherwise, the agent can be technically well-built but still give poor answers because the documentation is chaotic.

Teach the agent when not to answer

A helpful agent naturally tries to help. This is useful, but it can also become dangerous.

When the agent does not have enough information, it may still produce an answer that sounds reasonable. It may invent a theory, explain it confidently, and present it as if it were based on real data. For the user, this is worse than a simple error because the answer looks polished and trustworthy.

This often happens when the agent has a weak retrieval system, incomplete tool results, missing permissions, outdated documents, or no source of truth for the question. Instead of saying that the information is missing, the agent fills the gap with a plausible response.

A production system should not rely only on the model being “honest”. It should give the agent a clear way to recognize missing evidence.

For example, if the answer must be based on documents, the system should make it clear which documents were retrieved and whether they are relevant enough. If the answer depends on live product data, the agent should know whether the tool returned complete data, partial data, or an error. If the user asks about something outside the available knowledge, the agent should be allowed to say that it cannot verify the answer.

This is not a weakness. In many business scenarios, “I cannot confirm this from the available data” is a much better answer than a confident guess.

The practical rule is simple: the agent should not only be optimized to answer. It should also be designed to know when the available context is not enough.

A reliable agent is not the one that always sounds confident. A reliable agent is the one that can separate known facts from assumptions.

Product experience and scaling

Do not leave language handling to chance

Many products want the agent to answer in the user’s language. At first, this looks simple: add a prompt instruction that says, “Answer in the user’s language”.

In practice, this can be less stable than expected.

The agent may confuse similar languages. It may answer in a related but incorrect language. If the user writes in English, the situation can become even less predictable, especially in a multilingual product where the conversation history, retrieved documents, system instructions, and user profile may all contain different languages.

This is not always a model quality problem. Sometimes the model simply receives mixed signals.

A more reliable approach is to detect the user’s language explicitly before generating the main response. This can be done with a small, cheap model or a simple classification step that returns structured information, such as the full language name and language code.

For example, the product can detect that the current user message is Ukrainian and pass this to the main agent as dynamic context:

```
User language:
  Name: Ukrainian
  Code: uk
```

This gives the agent a clear instruction based on the current message instead of forcing it to infer the language from mixed context.

There is one architectural detail to keep in mind. Language detection is dynamic information. It may change from request to request, so it should not be mixed into the stable cached part of the prompt in a way that breaks prompt caching. Stable product instructions should remain stable. Dynamic language data should be added as a small request-specific block.

This makes the behavior more predictable without making the main prompt much larger.

The goal is simple: the user should not feel that the product randomly changes language. Language choice is part of the user experience, and it should be handled deliberately.

Do not lock the product into one model forever

Choosing one model for launch is normal. Locking the whole product architecture around that model forever is risky.

A production system should not change models randomly without control. But it also should not be painful to test or adopt a better model later.

Models change quickly. A new model may become cheaper, faster, or better for your specific use case. That is why the model should be a configuration choice, not the foundation of the entire product.

A good system allows the team to test new models on real tasks, compare quality, speed, cost, tool usage, and failure rates, and then migrate deliberately.

This does not mean the model should be changed every week. It means the team should not be technically locked into one vendor, one model name, or one response format.

The product should be stable enough not to break from accidental model changes, but flexible enough to benefit from better models when they appear.

More agents do not automatically mean a better product

Multi-agent systems sound advanced. They can be useful, but they do not automatically make the product better.

Every additional agent adds more instructions, more handoffs, more context passing, more tool calls, more latency, and more places for errors. This makes sense only when agents truly have different responsibilities.

For example, one agent may plan, another may search for information, and a third may verify the result. That can be useful when the workflow is complex and each role has a clear purpose.

But if several agents see too much context, duplicate work, and pass unstructured messages to each other, the system becomes not smarter, but more expensive and more complex.

In many cases, it is better to start with one well-designed agent, clear actions, good search, useful memory, and good metrics.

Multi-agent architecture should be added only when there is a specific problem it solves: different permissions, different domains, a complex review process, or the need to separate planning from

execution.

More agents should mean a better result, not just a prettier architecture diagram.

What this means in practice

A founder does not need to become an ML engineer to evaluate the quality of an agent architecture. But they should watch for several warning signs:

- The whole product depends on one giant prompt.
- All actions are always shown to the agent.
- Repeated input is not organized for caching.
- Several tools can perform similar actions, making the correct choice unclear.
- Broad tools allow the agent to change too much at once.
- Useful data is passed as noisy raw backend output.
- Risky actions depend only on prompt-based confirmation.
- Destructive or irreversible actions are not protected by product-level confirmation.
- Memory is just a long chat history.
- Search cannot handle both meaning and exact matches.
- Documents are split without preserving structure.
- The knowledge base contains outdated, duplicated, or conflicting information.
- The agent confidently answers even when it does not have enough data.
- Cost, latency, and token usage are not measured clearly.
- Detailed logs store too much sensitive information.
- Language handling depends only on a vague “answer in the user’s language” instruction.
- The architecture is tightly bound to one model or vendor.
- Multi-agent architecture is added without a clear reason.

These issues may look technical, but their impact is very practical. They affect margins, response speed, reliability, support workload, and the ability to improve the product after launch.

A good agent application is not just a smart model connected to a chat interface. It is a system that controls context, uses tools clearly, retrieves the right knowledge, tracks costs, protects sensitive data, and stays flexible enough to improve over time.

Technical decisions matter not because they are interesting on their own. They matter because they decide whether an agent product becomes a scalable business or remains an expensive demo.