in this course Lance Martin will teach you how to implement rag from scratch
Lance is a software engineer at Lang chain and Lang chain is one of the most common ways to implement rag Lance will
help you understand how to use rag to combine custom data with llms hi this is
Lance Martin I'm a software engineer at Lang chain I'm going to be giving a short course focused on rag or retrieval
augmented generation which is one of the most popular kind of ideas and in llms
today so really the motivation for this is that most of the world's data is
private um whereas llms are trained on publicly available data so you can kind
of see on the bottom on the x-axis the number of tokens using pre-training various llms so it kind of varies from
say 1.5 trillion tokens in the case of smaller models like 52 out to some very large number that we
actually don't know for proprietary models like GPT 4 CLA three but what's really interesting is
that the context window or the ability to feed external information into these
LMS is actually getting larger so about a year ago context windows were between 4 and 8,000 tokens you know that's like
maybe a dozen pages of text we've recently seen models all the way out to a million tokens which is thousands of
pages of text so while these llms are trained on large scale public data it's
increasingly feasible to feed them this huge mass of private data that they've never seen that private data can
be your kind of personal data it can be corporate data or you know other information that you want to pass to an
LM that's not natively in his training set and so this is kind of the main
motivation for rag it's really the idea that llms one are kind of the the center
of a new kind of operating system and two it's increasingly critical to be able to feed information
from external sources such as private data into llms for processing so that's
kind of the overarching motivation for Rag and now rag refers to retrieval
augmented generation and you can think of it in three very general steps there's a process of indexing of
external data so you can think about this as you know building a database for example um many companies already have
large scale databases in different forms they could be SQL DBS relational DBS um
they could be Vector Stores um or otherwise but the point is that
documents are indexed such that they can be retrieved based upon some heuristics
relative to an input like a question and those relevant documents can be passed to an llm and the llm can produce
answers that are grounded in that retrieved information so that's kind of the centerpiece or central idea behind
Rag and why it's really powerful technology because it's really uniting the the knowledge and processing
capacity of llms with large scale private external data source for which
most of the important data in the world still lives and in the following short videos

we're going to kind of build up a complete understanding of the rag landscape and we're going to be covering
a bunch of interesting papers and techniques that explain kind of how to do rag and I've really broken it down
into a few different sections so starting with a question on the left the first kind of section is what I call
query trans translation so this captures a bunch of different methods to take a question from a user and modify it in
some way to make it better suited for retrieval from you know one of these indexes we've talked
about that can use methods like query writing it can be decomposing the query
into you know constituent sub questions then there's a question of routing so taking that decomposed a
Rewritten question and routing it to the right place you might have multiple Vector stores a relational DB graph DB
and a vector store so it's the challenge of getting a question to the right Source then there's a there's kind of
the challenge of query construction which is basically taking natural language and converting it into the DSL
necessary for whatever data source you want to work with a classic example here is text a SQL which is kind of a very
kind of well studied process but text a cipher for graph DV is very interesting
text to metadata filters for Vector DBS is also a very big area of
study um then there's indexing so that's the process of taking your documents and
processing them in some way so they can be easily retrieved and there's a bunch of
techniques for that we'll talk through we'll talk through different
embedding methods we'll talk about different indexing strategies after
retrieval there are different techniques to rerank or filter retrieve documents
um and then finally we'll talk about generation and kind of an interesting new set of methods to do what we might
call as active rag so in that retrieval or generation stage grade documents
grade answers um grade for relevance to the question grade for faithfulness to
the documents I.E check for hallucinations and if either fail feedback uh re- retrieve or rewrite the
question uh regenerate the qu regenerate the answer and so forth so there's a really interesting set of methods we're
going to talk through that cover that like retrieval and generation with
feedback and you know in terms of General outline we'll cover the basics first it'll go through indexing
retrieval and generation kind of in the Bare Bones and then we'll talk through more advanced techniques that we just
saw on the prior slide career Transformations routing uh construction and so forth hi this is Lance from Lang

chain this the second video in our series rack from scratch focused on indexing
so in the past video you saw the main kind of overall components of rag pipelines indexing retrieval and

generation and here we're going to kind of Deep dive on indexing and give like just a quick overview of it so the first
aspect of indexing is we have some external documents that we actually want to load and put into what we're trying
to call Retriever and the goal of this retriever is simply given an input question I want to fish out doents that
are related to my question in some way now the way to establish that
relationship or relevance or similarity is typically done using some kind of numerical representation of documents
and the reason is that it's very easy to compare vectors for example of numbers
uh relative to you know just free form text and so a lot of approaches have
been a developed over the years to take text documents and compress them down
into a numerical rep presentation that then can be very easily searched now there's a few ways to do
that so Google and others came up with many interesting statistical methods where you take a document you look at
the frequency of words and you build what they call sparse vectors such that the vector locations are you know a
large vocabulary of possible words each value represents the number of occurrences of that particular word and
it's sparse because there's of course many zeros it's a very large vocabulary relative to what's present in the
document and there's very good search methods over this this type of numerical representation now a bit more recently
uh embedding methods that are machine learned so you take a document and you build a compressed fixed length
representation of that document um have been developed with
correspondingly very strong search methods over embeddings um so the intuition here is
that we take documents and we typically split them because embedding models
actually have limited context windows so you know on the order of maybe 512 tokens up to 8,000 tokens or Beyond but
they're not infinitely large so documents are split and each document is compressed into a vector and that Vector
captures a semantic meaning of the document itself the vectors are indexed
questions can be embedded in the exactly same way and then numerical kind of
comparison in some form you know using very different types of methods can be performed on these vectors to fish out
relevant documents relative to my question um and let's just do a quick
code walk through on some of these points so I have my notebook here I've
installed here um now I've set a few API keys for lsmith which are very useful
for tracing which we'll see shortly um previously I walked through this this kind of quick start that just
showed overall how to lay out these rag pipelines and here what I'll do is I'll Deep dive a little bit more on indexing
and I'm going to take a question and a document and first I'm just going to
compute the number of tokens in for example the question and this is interesting because embedding models in

llms more generally operate on tokens and so it's kind of nice to understand how large the documents are that I'm
trying to feed in in this case it's obviously a very small in this case question now I'm going to specify open
eye embeddings I specify an embedding model here and I just say embed embed
query I can pass my question my document and what you can see here is that runs
and this is mapped to now a vector of length 1536 and that fixed length Vector
representation will be computed for both documents and really for any document so
you're always is kind of computing this fix length Vector that encodes the semantics of the text that you've passed
now I can do things like cosine similarity to compare them and as we'll see here I can load
some documents this is just like we saw previously I can split them and I can index them here just like
we did before but we can see under the hood really what we're doing is we're taking each split we're embedding it
using open eye embeddings into this this kind of this Vector representation and that's stored with a link to the rod
document itself in our Vector store and next we'll see how to actually do retrieval using this Vector store hi

this is Lance from Lang chain and this is the third video in our series rag from scratch building up a lot of the
motivations for rag uh from the very basic components um so we're going to be
talking about retrieval today in the last two uh short videos I outlined
indexing and gave kind of an overview of this flow which starts with indexing of our documents retrieval of documents
relevant to our question and then generation of answers based on the retriev documents and so we saw that the
indexing process basically makes documents easy to retrieve and it goes
through a flow that basically looks like you take our documents you split them in some way into these smaller chunks that
can be easily embedded um those embeddings are then numerical representations of those documents that
are easily searchable and they're stored in an index when given a question that's also
embedded the index performs a similarity search and returns splits that are relevant to the
question now if we dig a little bit more under the hood we can think about it like this if we take a document and
embed it let's imagine that embedding just had three dimensions so you know each document is projected into some
point in this 3D space now the point is that the location in space is determined by the semantic
meaning or content in that document so to follow that then documents in similar
locations in space contain similar semantic information and this very
simple idea is really the Cornerstone for a lot of search and retrieval methods that you'll see with modern Vector stores so in particular we take
our documents we embed them into this in this case a toy 3D space we take our question do the

same we can then do a search like a local neighborhood search you can think about in this 3D space around our
question to say hey what documents are nearby and these nearby neighbors are
then retrieved because they can they have similar semantics relative to our
question and that's really what's going on here so again we took our documents
we split them we embed them and now they exist in this high dimensional space we've taken our question embedded it
projected in that same space and we just do a search around the question from nearby documents and grab ones that are
close and we can pick some number we can say we want one or two or three or n documents close to my question in this
embedding space and there's a lot of really interesting methods that implement this very effectively I I link
one here um and we have a lot of really nice
uh Integrations to play with this general idea so many different embedding models many different indexes lots of
document loaders um and lots of Splitters that can be kind of recombined to test different ways of doing this
kind of indexing or retrieval um so now I'll show a bit of a code walkth through so here we defined
um we kind of had walked through this previously this is our notebook we've installed a few packages we've set a few
environment variables using lsmith and we showed this previously this is just an overview showing how to run rag
like kind of end to end in the last uh short talk we went through indexing um and what I'm going to do
very simply is I'm just going to reload our documents so now I have our documents
I'm going to resplit them and we saw before how we can build our
index now here let's actually do the same thing but in the slide we actually showed kind of that notion of search in
that 3D space and a nice parameter to think about in building your your retriever is
K so K tells you the number of nearby neighbors to fetch when you do that retrieval process and we talked about
you know in that 3D space do I want one nearby neighbor or two or three so here we can specify k equals 1 for example
now we're building our index so we're taking every split embedding it storing it now what's nice is I asked a a
question what is Task decomposition this is related to the blog post and I'm going to run get relevant documents so I
run that and now how many documents do I get back I get one as expected based upon k equals 1 so this retrieve
document should be related to my question now I can go to lsmith and we
can open it up and we can look at our Retriever and we can see here was our question here's the one document we got
back and okay so that makes sense this document pertains to task ke
decomposition in particular and it kind of lays out a number of different approaches that can be used to do that

this all kind of makes sense and this shows kind of in practice how you can implement this this NE this kind of KNN
or k nearest neighbor search uh really easily uh just using a few lines of code
and next we're going to talk about generation thanks hey this is Lance from Lang chain

this is the fourth uh short video in our rack from scratch series that's going to be focused on generation now in the past few videos we walked through the general flow uh for
kind of basic rag starting with indexing Fall by retrieval then
generation of an answer based upon the documents that we retrieved that are relevant to our question this is kind of
the the very basic flow now an important consideration in
generation is really what's happening is we're taking the documents you retrieve and we're stuffing them into the llm
context window so if we kind of walk back through the process we take documents we split them for convenience
or embedding we then embed each split and we store that in a vector store as
this kind of easily searchable numerical representation or vector and we take a question embed it to produce a similar
kind of numerical representation we can then search for example using something like KN andn in this kind of dimensional
space for documents that are similar to our question based on their proximity or
location in this space in this case you can see 3D is a toy kind of toy
example now we've recovered relevant splits to our question we pack those into the context window and we produce
our answer now this introduces the notion of a prompt so the prompt is kind of a you
can think have a placeholder that has for example you know in our case B keys
so those keys can be like context and question so they basically are like
buckets that we're going to take those retrieve documents and Slot them in we're going to take our question and
also slot it in and if you kind of walk through this flow you can kind of see that we can build like a dictionary from
our retrieve documents and from our question and then we can basically populate our prompt template with the
values from the dict and then becomes a prompt value which can be passed to llm
like a chat model resulting in chat messages which we then parse into a string and get our answer so that's like
the basic workflow that we're going to see and let's just walk through that in code very quickly to kind of give you
like a Hands-On intuition so we had our notebook we walk through previously install a few packages I'm setting a few
lsmith environment variables we'll see it's it's nice for uh kind of observing and debugging our
traces um previously we did this quick start we're going to skip that over
um and what I will do is I'm going to build our retriever so again I'm going
to take documents and load them uh and then I'm going to split them here we've kind of done this previously so I'll go
through this kind of quickly and then we're going to embed them and store them in our index
so now we have this retriever object here now I'm going to

jump down here now here's where it's kind of fun this is the generation bit and you can see here I'm defining
something new this is a prompt template and what my prompt template is something really simple it's just going to say
answer the following question based on this context it's going to have this context variable and a question so now
I'm building my prompt so great now I have this prompt let's define an llm I'll choose
35 now this introdu the notion of a chain so in Lang chain we have an expression language called L Cel Lang
chain expression language which lets you really easily compose things like prompts LMS parsers retrievers and other
things but the very simple kind of you know example here is just let's just take our prompt which you defined right
here and connect it to an LM which you defined right here into this chain so there's our chain now all we're doing is
we're invoking that chain so every L expression language chain has a few common methods like invoke bat stream in
this case we just invoke it with a dict so context and question that maps to the
expected Keys here in our template and so if we run invoke what we see is
it's just going to execute that chain and we get our answer now if we zoom over to Langs Smith we should see that
it's been populated so yeah we see a very simple runable sequence here was our
document um and here's our output and here is our prompt answer the following
question based on the context here's the document we passed in here is the question and then we get our answer so
that's pretty nice um now there's a lot of other options for rag prompts I'll
pull one in from our prompt tub this one's like kind of a popular prompt so it just like has a little bit more
detail but you know it's the main the main intuition is the same um you're
passing in documents you're asking them to reason about the documents given a question produce an answer and now here
I'm going to find a rag chain which will automatically do the retrieval for us and all I have to do is specify here's
my retriever which we defined before here's our question we which we invoke with the question gets passed
through to the key question in our dict and it automatically will trigger the
retriever which will return documents which get passed into our context so it's exactly what we did up here except
before we did this manually and now um this is all kind of automated for
us we pass that dick which is autop populated into our prompt llm out to parser now
let invoke it and that should all just run and great we get an answer and we
can look at the trace and we can see everything that happened so we can see our retriever was
run these documents were retrieved they get passed into our
LM and we get our final answer so this kind of the end of our overview um where
we talked about I'll go back to the slide here quickly we talked about indexing retrieval and now

generation and follow-up short videos we'll kind of dig into some of the more com complex or detailed themes that
address some limitations that can arise in this very simple pipeline thanks hi my from Lang chain over the

next few videos we're going to be talking about career translation um and in this first video
we're going to cover the topic of multi-query so query translation sits kind of at the
first stage of an advanced rag Pipeline and the goal of career translation is
really to take an input user question and to translate in some way in order to
improve retrieval so the problem statement is pretty intuitive user queries um can be
ambiguous and if the query is poorly written because we're typically doing some kind of semantic similarity search
between the query and our documents if the query is poorly written or ill opposed we won't retrieve the proper
documents from our index so there's a few approaches to
attack this problem and you can kind of group them in a few different ways so
here's one way I like to think about it a few approaches has involveed query rewriting so taking a query and
reframing it like writing from a different perspective um and that's what we're going to talk about a little bit
here in depth using approaches like multi-query or rag Fusion which we'll talk about in the next video you can
also do things like take a question and break it down to make it less abstract like into sub questions and there's a
bunch of interesting papers focused on that like least to most from Google you can also take the opposite
approach of take a question to make it more abstract uh and there's actually approach we're going to talk about later
in a future video called stepback prompting that focuses on like kind of higher a higher level question from the
input so the intuition though for this multier approach is we're taking a
question and we're going to break it down into a few differently worded questions uh from different
perspectives and the intuition here is simply that um it is possible that the
way a question is initially worded once embedded it is not well aligned or in
close proximity in this High dimensional embedding space to a document that we want to R that's actually related so the
thinking is that by kind of rewriting it in a few different ways you actually increase the likel of actually
retrieving the document that you really want to um because of nuances in the way
that documents and questions are embedded this kind of more shotgun approach of taking a question Fanning it
out into a few different perspectives May improve and increase the reliability of retrieval that's like the intuition
really um and of course we can com combine this with retrieval so we can
take our our kind of fan out questions do retrieval on each one and combine them in some way and perform rag so
that's kind of the overview and now let's what let's go over to um our code

so this is a notebook and we're going to share all this um we're just installing a few packages we're setting a lsmith API Keys which we'll see why that's quite useful here shortly there's our diagram now
first I'm going to Index this blog post on agents I'm going to split it um well
I'm going to load it I'm going to split it and then I'm going to index it in chroma locally so this is a vector store
we've done this previously so now I have my index defined so here is where I'm defining my prompt for multiquery which
is your your assistant your task is to basically reframe this question into a few different sub questions um so there's our prompt um right here we'll pass that to
an llm part it um into a string and then split the string by new lines and so
we'll get a list of questions out of this chain that's really all we're doing here now all we're doing is here's a
sample input question there's our generate queries chain which we defined we're going to take that list and then
simply apply each question to retriever so we'll do retrieval per question and
this little function here is just going to take the unique Union of documents uh across all those retrievals so let's run
this and see what happens so we're going to run this and we're going to get some set of questions uh or documents back so
let's go to Langs Smith now we can actually see what happened under the hood so here's the key
point we ran our initial chain to generate a set of of reframed questions
from our input and here was that prompt and here is that set of questions that we generated now what happened is for
every one of those questions we did an independent retrieval that's what we're showing here so that's kind of the first
step which is great now I can go back to the notebook and we can show this working end to end so now we're going to
take that retrieval chain we'll pass it into context of our final rag prompt we'll also pass through the question
we'll pass that to our rag prompt here pass it to an LM and then Pary output
now let's let's kind of see how that works so again that's okay there it is
so let's actually go into langth and see what happened under the hood so this was our final chain so this is great we took
our input question we broke it out to these like five rephrase questions for
every one of those we did a retrieval that's all great we then took the unique Union of documents and you can see in
our final llm prompt answer the following cont following question based on the context this is the final set of
unique documents that we retrieved from all of our sub questions um here's our initial question
there's our answer so that kind of shows you how you can set this up really easily how you can use l Smith to kind of investigate what's going on and in
particular use l Smith to investigate those intermediate questions that you generate in that like kind of question
generation phase and in a future talks we're going to go through um some of these other methods that we kind of

introduced at the start of this one thank you last L chain this is the second

video of our Deep dive on query translation in our rag from scratch series focused on a method called rag
Fusion so as we kind of showed before career translation you can think of as the first stage in an advanced rag
pipeline we're taking an input user question and We're translating it some way in order to improve
retrievable now we showed this General mapping of approaches previously so
again you have kind of like rewriting so you can take a question and like kind of break it down into uh differently worded
are different different perspectives of the same question so that's kind of rewriting there's sub questions where
you take a question break it down into smaller problems solve each one independently and then there step back
where you take a question and kind of go more abstract where you kind of ask a higher level question as a precondition
to answer the user question so those are the approaches and we're going to dig into one of the particular approaches
for rewriting called rat Fusion now this is really similar to what we just saw
with multiquery the difference being we actually apply a a kind of a clever rank ranking step of
our retriev documents um which you call reciprocal rank Fusion that's really the only difference the the input stage of
taking a question breaking it out into a few kind of differently worded questions
retrieval on each one is all the same and we're going to see that in the code here shortly so let's just hop over
there and then look at this so again here is a notebook that we introduced
previously here's the packages we've installed we've set a few API keys for lsmith which we see why is quite
useful um and you can kind of go down here to a rag Fusion
section and the first thing you'll note is what our prompt is so it looks really similar to The Prompt we just saw with
multiquery and simply your helpful assistant that generates multiple search queries based upon user input and here's
the question output for queries so let's define our prompt and here was our query
Generation chain again this looks a lot like we just saw we take our prompt Plum that into an llm and then basically
parse by new lines and that'll basically split out these questions into a list
that's all it's going to happen here so that's cool now here's where the novelty comes
in each time we do retrieval from one of those questions we're going to get back
a list of documents from our Retriever and so we do it over that we generate
four questions here based on our prompt we do the over four questions well like a list of lists
basically now reciprocal rank Fusion is really well suited for this exact problem we want to take this list to
list and build a single Consolidated list and really all that's going on is it's looking at the documents in each
list and kind of aggregating them into a final output ranking um and that's

really the intuition around what's happening
here um so let's go ahead
and so
let's so let's go ahead and look at that in some detail so we can see we
run retrieval that's great now let's go over to Lang Smith and have a look at what's
going on here so we can see that here was our prompt to your helpful assistant
that generates multiple search queries based on a single input and here is our search
queries and then here are our
four retrievals so that's that's really good so we know that all is
working um and then those retrievals simply went into this rank
function and our correspondingly ranked to a final list of six unique rank
documents that's really all we did so let's actually put that all
together into an a full rag chain that's going to run
retrieval return that final list of rank documents and pass it to our context
pass through our question send that to a rag prompt pass it to an LM parse it to
an output and let's run all that together and see that working cool so there's our final
answer now let's have a look in lsmith we can see here was our four questions
here's our retrievals and then our final rag prompt plumed through the final list
of ranked six questions which we can see laid out here and our final answer so
this can be really convenient particularly if we're operating across like maybe different Vector
stores uh or
we want to do like retrieval across a large number of of kind of differently worded questions
this reciprocal rank
Fusion step is really nice um for example if we wanted to only take the top three documents
or something um it
can be really nice to build that Consolidated ranking across all these independent retrievals
then pass that to
for the final generation so that's really the intuition about what's happening here thanks

hi this is Lance from Lang chain this is our third video focused on query translation in the rag
from scratch
series and we're going to be talking about decomposition so query translation in
general is a set of approaches that sits kind of towards the front of this overall rag Pipeline
and the objective
is to modify or rewrite or otherwise decompose an input question from a user
in order improve retrieval so we can talk through some of
these approaches previously in particular various ways to do query writing like rag fusion
and multiquery
there's a separate set of techniques that become pretty popular and are really interesting for
certain problems
which we might call like kind of breaking down or decomposing an input question into a set
of sub
questions um so some of the papers here that are are pretty cool are for example
this work from Google um and the objective really is
first to take an input question and decompose it into a set of sub problems
so this particular example from the paper was the problem of um last letter
concatenation and so it took the inut question of three words think machine

learning and broke it down into three sub problems think think machine think machine learning as the third sub
problem and then you can see in this bottom panel it solves each one individually so it shows for example in
green solving the problem think machine where you can catenate the last letter of k with the last letter of machine or
last letter think K less machine e can concatenate those to K and then for the
overall problem taking that solution and then and basically building on it to get
the overall solution of keg so that's kind of one concept of decomposing into sub problems solving them
sequentially now a related work called IRC or in leap retrieval combines
retrieval with Chain of Thought reasoning and so you can kind of put these together into one approach which
you can think of as kind of dynamically retrieval um to solve a set of sub
problems kind of that retrieval kind of interleaving with Chain of Thought as noted in the second paper and a set of
decomposed questions based on your initial question from the first work from Google so really the idea here is
we're taking one sub question we're answering it we're taking that answer and using it to help answer the second
sub question and so forth so let's actually just walk through this in code to show how this might
work so this is The Notebook we've been working with from some of the other uh videos you can see we already have a
retriever to find uh up here at the top and what we're going to do
is we're first going to find a prompt that's basically going to say given an
input question let's break it down to set of sub problems or sub question which can be solved individually so we
can do that and this blog post is focused on agents so let's ask a question about what are the main
components of an LM powerered autonomous agent system so let's run this and
see what the decomposed questions are so you can see the decomposed questions are what is LM technology how does it work
um what are components and then how the components interact so it's kind of a sane way to kind of break down this
problem into a few sub problems which you might attack individually now here's
where um we Define a prompt that very simply is going to take our question
we'll take any prior questions we've answered and we'll take our retrieval and basically just combine them and we
can Define this very simple chain um actually let's go back and make sure retriever is defined up at the
top so now we are building our retriever good we have that now so we
can go back down here and let's run this so now we are
running and what's happening is we're trying to solve each of these questions
individually using retrieval and using any prior question answers so okay very
good looks like that's been done and we can see here's our answer now let's go over to
langth and actually see what

happened under the hood so here's what's kind of of interesting and helpful to see for the first question so here's our
first one it looks like it just does retrieval which is we expect and then it uses that to answer this initial
question now for the second question should be a little bit more interesting because if you look at our prompt here's
our question now here is our background available question answer pair so this
was the answer question answer pair from the first question which we add to our prompt and then here's the retrieval for
this particular question so we're kind of building up up the solution because we're pending the question answer pair
from question one and then likewise with question three it should combine all of
that so we can look at here here's our question here's question one here's question two great now here's additional
retrieval related to this particular question and we get our final answer so that's like a really nice way you can
kind of build up Solutions um using this kind of interleaved uh retrieval and
concatenating question answer pairs I do want to mention very briefly that we can
also take a different approach where we can just answer these all individually and then just concatenate all those
answers to produce a final answer and I'll show that really quickly here um
it's like a little bit less interesting maybe because you're not using answers from each uh question to inform the next
one you're just answering them all in parallel this might be better for cases where it's not really like a sub
question decomposition but maybe it's like like a set of set of several in independent questions whose answers
don't depend on each other that might be relevant for some problems um and we can go ahead and run
okay so this ran as well we can look at our trace and in this
case um yeah we can see that this actually just kind of concatenates all of our QA pairs to produce the final
answer so this gives you a sense for how you can use quer decomposition employ IDE IDE from uh from two different
papers that are pretty cool thanks hi this is Lance from Lang chain this is

the fourth video uh in our Deep dive on queer translation in the rag from
scratch series and we're going to be focused on step back prompting so queer translation as we
said in some of the prior videos kind of sits at the the kind of first stage of
kind of a a a rag pipeline or flow and the main aim is to take an question and
to translate it or modify in such a way that it improves retrieval now we talked through a few
different ways to approach this problem so one General approach involves rewriting a question and we talk about
two ways to do that rag fusion multiquery and again this is this is really about taking a question and
modifying it to capture a few different perspectives um which may improve the retrieval

process now another approach is to take a question and kind of make it less abstract like break it down into sub
questions um and then solve each of those independently so that's what we saw with like least to most prompting um
and a bunch of other variants kind of in that in that vein of sub problem solving
and then consolidating those Solutions into a final answer now a different approach
presented um by again Google as well is stepback prompting so stepback prompting
kind of takes the the the opposite approach where it tries to ask a more abstract question so the paper talks a
lot about um using F shot prompting to produce what they call the
stepback or more abstract questions and the way it does it is it provides a number of examples of stepb back questions given your original question so like this is
like this is for example they like for prompt temp you're an expert World Knowledge I asked you a question your response should be
comprehensive not contradict with the following um and this is kind of where you provide your like original and then
step back so here's like some example um questions so like um
like uh at year saw the creation of the region where the country is located
which region of the country um is the county of of herir
related um Janell was born in what country what is janell's personal
history so that that's maybe a more intuitive example so it's like you ask a very specific question about like the country someone's born the more abstract
question is like just give me the general history of this individual without worrying about that particular
um more specific question um so let's actually just walk through how this can
be done in practice um so again here's kind of like a a diagram of uh the
various approaches um from less abstraction to more abstraction now here is where we're formulating our prompt using a few of the few shot examples from the paper um so again like input um yeah
something about like the police perform wful arrests and what what camp members of the police do so like it it basically
gives the model a few examples um we basically formulate this into a prompt
that's really all going on here again we we repeat um this overall prompt which
we saw from the paper your expert World Knowledge your test is to step back and paraphrase a question generate more a
generic step back question which is easier to answer here are some examples so it's like a very intuitive prompt so
okay let's start with the question what is Task composition for llm agents and
we're going to say generate stack question okay so this is pretty intuitive right what is a process of task compos I so like not worrying as
much about agents but what is that process of task composition in general and then hopefully that can be
independently um retrieved we we can independently retrieve documents related to the stepb back question and in
addition retrieve documents related to the the actual question and combine those to produce kind of final answer so
that's really all that's going on um and here's the response template where we're Plumbing in the stepback context and our

question context and so what we're going to do here is we're going to take our input question and perform retrieval on
that we're also going to generate our stepb back question and perform retrieval on that we're going to plumb
those into the prompt as here's our very here's our basically uh our prompt Keys
normal question step back question um and our overall question again we
formulate those as a dict we Plum those into our response prompt um and then we
go ahead and attempt to answer our overall question so we're going to run that that's
running and okay we have our answer now I want to hop over to Langs Smith and
attempt to show you um kind of what that looked like under the hood so let's see
let's like go into each of these steps so here was our prompt right you're an expert World Knowledge your test to to
step back and paraph as a question um so um here were our few shot prompts
and this was our this was our uh stepb question so what is the process of task
composition um good from the input what is Tas composition for LM agents we
perform retrieval on both what is process composition uh and what is for LM agents
we perform both retrievals we then populate our prompt with both
uh original question answer and then here's the context retrieve from both the question and the stepb back question
here was our final answer so again this is kind of a nice technique um probably depends on a lot of the types of like
the type of domain you want to perform retrieval on um but in some domains
where for example there's a lot of kind of conceptual knowledge that underpins questions you expect users to ask this
stepback approach could be really convenient to automatically formulate a
higher level question um to for example try to improve retrieval I can imagine if you're working with like kind of
textbooks or like technical documentation where you make independent chapters focused on more highlevel kind
of like Concepts and then other chapters on like more detailed uh like
implementations this kind of like stepb back approach and independent retrieval could be really helpful thanks hi this

is Lance from Lang chain this is the fifth video focused on queer translation in our rack from scratch
series we're going to be talking about a technique called hide so again queer translation sits
kind of at the front of the overall rag flow um and the objective is to take an input question and translate it in some
way that improves retrieval now hide is an interesting
approach that takes advantage of a very simple idea the basic rag flow takes a
question and embeds it takes a document and embeds it and looks for similarity between an embedded document and
embedded question but questions and documents are very different text objects so documents can be like very
large chunks taken from dense um Publications or other sources whereas
questions are short kind of tur potentially ill worded from users and the intuition behind hide is take
questions and map them into document space using a hypothetical document or

by generating a hypothetical document um that's the basic intuition and the idea
kind of shown here visually is that in principle for certain cases a hypothetical document is closer to a
desired document you actually want to retrieve in this you know High dimensional embedding space than the
sparse raw input question itself so again it's just kind of means of trans translating raw questions into these
hypothetical documents that are better suited for retrieval so let's actually do a Code
walkthrough to see how this works and it's actually pretty easy to implement which is really nice so first we're just
starting with a prompt and we're using the same notebook that we've used for prior videos we have a blog post on
agents r index um so what we're going to do is Define a prompt to generate a
hypothetical documents in this case we'll say write a write a paper passage uh to answer a given question so let's
just run this and see what happens again we're taking our prompt piping it to to open Ai chck gpte and then using string
Opa parer and so here's a hypothetical document section related to our question
okay and this is derived of course lm's kind of embedded uh kind of World Knowledge which is you know a sane place
to generate hypothetical documents now let's now take that hypothetical
document and basically we're going to pipe that into a retriever so this means we're going to fetch documents from our
index related to this hypothetical document that's been embedded and you can see we get a few qu a few retrieved
uh chunks that are related to uh this hypothetical document that's all we've
done um and then let's take the final step where we take those retrieve
documents here which we defined and our question we're going to pipe that into this rag prompt and then
we're going to run our kind of rag chain right here which you've seen before and we get our answer so that's really it we
can go to lsmith and we can actually look at what happened um so here for
example this was our final um rag prompt
answer the following question based on this context and here is the retrieve documents that we passed in so that
part's kind of straightforward we can also look at um okay this is our
retrieval okay now this is this is actually what we we generated a
hypothetical document here um okay so this is our hypothetical document so
we've run chat open AI we generated this passage with our hypothetical document and then we've run
retrieval here so this is basically showing hypothetical document generation followed by retrieval um so again here
was our passage which we passed in and then here's our retrieve documents from the retriever which are related to the
passage content so again in this particular index case it's possible that the input question was sufficient to
retrieve these documents in fact given prior examples uh I know that some of these same documents are indeed

retrieved just from the raw question but in other context it may not be the case so folks have reported nice performance
using Hyde uh for certain domains and the Really convenient thing is that you
can take this this document generation prompt you can tune this arbitrarily for
your domain of Interest so it's absolutely worth experimenting with it's a it's a need approach uh that can
overcome some of the challenges with retrieval uh thanks very much hi this is

Lance from Lang chain this is the 10th video in our rack from scratch series focused on routing so we talk through query translation which is the process of taking a question and translating in
some way it could be decomposing it using stepback prompting or otherwise but the idea here was take our question
change it into a form that's better suited for retrieval now routing is the next step which is basically routing
that potentially decomposed question to the right source and in many cases that could be a different database so let's
say in this toy example we have a vector store a relational DB and a graph DB the
what we redo with routing is we simply route the question based upon the cont of the question to the relevant data
source so there's a few different ways to do that one is what we call logical routing in this case we basically give
an llm knowledge of the various data sources that we have at our disposal and
we let the llm kind of Reason about which one to apply the question to so
it's kind of like the the LM is applying some logic to determine you which which data sour for example to to use
alternatively you can use semantic routing which is where we take a question we embed it and for example we
embed prompts we then compute the similarity between our question and those prompts and then we choose a
prompt based upon the similarity so the general idea is in our diagram we talk about routing to for example a different
database but it can be very general can be routing to different prompt it can be you know really arbitrarily taking this
question and sending it at different places be at different prompts be at different Vector stores so let's walk through the code a
little bit so you can see just like before we've done a few pip installs we set up lsmith and let's talk through uh
logical routing first so so in this toy example let's say we had for example uh
three different docs like we had python docs we had JS docs we had goang docs
what we want to do is take a question route it to one of those three so what we're actually doing is we're setting up
a data model which is basically going to U be bound to our llm and allow the llm
to Output one of these three options as a structured object so you really think
about this as like classification classification plus function calling to produce a structured output which is constrained to these three possibilities so the way we do that is
let's just zoom in here a little bit we can Define like a structured object that we want to get out from our llm like in

this case we want for example you know one of these three data sources to be output we can take this and we can
actually convert it into open like open for example function schema and then we actually pass that in and
bind it to our llm so what happens is we ask a question our llm invokes this
function on the output to produce an output that adheres to the schema that we specify so in this case for example
um we output like you know in this toy example let's say we wanted like you know an output to be data source Vector
store or SQL database the output will contain a data source object and it'll be you know one of the options we
specify as a Json string we also instantiate a parser from this object to
parse that Json string to an output like a pantic object for example so that's
just one toy example and let's show one up here so in this case again we had our three doc sources um we bind that to our
llm so you can see we do with structured output basically under the hood that's
taking that object definition turning into function schema and binding that function schema to our llm and we call
our prompt you're an expert at routing a user question based on you know programming language um that user
referring to so let's define our router here now what we're going to do is we'll
ask a question that is python code so we'll call that and now it's done and
you see the object we get out is indeed it's a route query object so it's exactly it aderes to this data model
we've set up and in this case it's it's it's correct so it's calling this python
doc so you can we can extract that right here as a string now once we have this
you can really easily set up like a route so this could be like our full chain where we take this router we
should defined here and then this choose route function can basically take that
output and do something with it so for example if python docs this could then apply the question to like a retriever
full of python information uh or JS same thing so this is where you would hook
basically that question up to different chains that are like you know retriever chain one for python retriever chain two
for JS and so forth so this is kind of like the routing mechanism but this is really doing the heavy lifting of taking
an input question and turning into a structured object that restricts the output to one of a few output types that
we care about in our like routing problem so that's really kind of the way this all hooks
together now semantic outing is actually maybe even a little bit more straightforward based on what we've seen
previously so in that case let's say we have two prompts we have a physics prompt we have a math
prompt we can embed those prompts no problem we do that here now let's say we
have an input question from a user like in this case what is a black hole we pass that through we then apply this
runnable Lambda function which is defined right here what we're doing here is we're embedding the question we're

Computing similarity between the question and the prompts uh we're taking the most similar and then we're
basically choosing the prompt based on that similarity and you can see let's run that and try it
out and we're using the physics prompt and there we go black holes region and space so that just shows you kind of how
you can use semantic routing uh to basically embed a question embed for
example various prompts pick the prompt based on sematic similarity so that really gives you just two ways to do
routing one is logical routing with function in uh can be used very generally in this case we applied it to
like different coding languages but imagine these could be swapped out for like you know my python uh my like
vector store versus My Graph DB versus my relational DB and you could just very
simply have some description of what each is and you know then not only will the llm do reasoning but it'll also
return an object uh that can be parsed very cleanly to produce like one of a
few very specific types which then you can reason over like we did here in your routing function so that kind of gives
you the general idea and these are really very useful tools and I encourage you to experiment with them
thanks hi this is Lance from Lang chain this is the 11th part of our rag from scratch video series focused on query

construction so we previously talked through uh query translation which is the process of taking a question and
converting it or translating it into a question that's better optimized for retrieval then we talked about routing
which is the process of going taking that question routing it to the right Source be it a given Vector store graph
DB um or SQL DB for example now we're going to talk about the process of query construction which is basically taking
natural language and converting it into particular domain specific language uh for one of these sources now we're going
to talk specifically about the process of going from natural language to uh meditated filters for Vector
Stores um the problem statement is basically this let's imagine we had an index of Lang Chain video transcripts um
you might want to ask a question give me you know or find find me videos on chat Lang chain published after 2024 for
example um the the process of query structuring basically converts this
natural language question into a structured query that can be applied to the metadata uh filters on your vector
store so most Vector stores will have some kind of meditative filters that can do kind of structur querying on top of
uh the chunks that are indexed um so for example this type of query will retrieve all chunks uh that talk about the topic
of chat Lang chain uh published after the date 2024 that's kind of the problem

statement and to do this we're going to use function calling um in this case you can use for example open AI or other
providers to do that and we're going to do is at a high level take the metadata
fields that are present in our Vector store and divide them to the model as kind of information and the model then
can take those and produce queries that adhere to the schema provided um and
then we can parse those out to a structured object like a identic object which again which can then be used in
search so that's kind of the problem statement and let's actually walk through code um so here's our notebook which
we've kind of gone through previously and I'll just show you as an example let's take a example YouTube video and
let's look at the metadata that you get with the transcript so you can see you get stuff like description uh URL um
yeah publish date length things like that now let's say we had an index that had um basically a that had a number of
different metadata fields and filters uh that allowed us to do range filtering on
like view count publication date the video length um or unstructured search on contents and title so those are kind
of like the imagine we had an index that had uh those kind of filters available
to us what we can do is capture that information about the available filters in an object so we're calling that this
tutorial search object kind of encapsulates that information about the available searches that we can do and so
we basically enumerate it here content search and title search or semantic searches that can be done over those
fields um and then these filters then are various types of structure searches
we can do on like the length um The View count and so forth and so we can just
kind of build that object now we can set this up really easily with a basic simple prompt that says you know you're
an expert can bring natural language into database queries you have access to the database tutorial videos um given a
question return a database query optimize retrieval so that's kind of it now here's the key point though when you
call this LM with structured output you're binding this pantic object which contains all the information about our
index to the llm which is exactly what we talked about previously it's really this process right here you're taking
this object you're converting it to a function schema for example open AI you're binding that to your model and
then you're going to be able to get um structured object out versus a Json
string from a natural language question which can then be parsed into a pantic
object which you get out so that's really the flow and it's taking advantage of function calling as we said
so if we go back down we set up our query analyzer chain right here now let's try to run that just on a on a
purely semantic input so rag from scratch let's run that and you can see this just does like a Content search and

a title search that's exactly what you would expect now if we pass a question that includes like a date filter let's
just see if that would work and there we go so you kind of still get
that semantic search um but you also get um search over for example publish date
earliest and latest publish date kind of as as you would expect let's try another one here so videos focus on the topic of
chat Lang chain they're published before 2024 this is just kind of a rewrite of this question in slightly different way
using a different date filter and then you can see we can get we get content search title search and then we can get
kind of a date search so this is a very general strategy that can be applied kind of broadly to um different kinds of
querying you want to do it's really the process of going from an unstructured input to a structured query object out
following an arbitrary schema that you provide and so as noted really this
whole thing we created here this tutorial search is based upon the specifics of our Vector store of
interest and if you want to learn more about this I link to some documentation here that talks a lot about different uh
types of of Integrations we have with different Vector store providers to do exactly this so it's a very useful trick
um it allows you to do kind of query uh uh say metadata filter filtering on the
fly from a natural language question it's a very convenient trick uh that works with many different Vector DBS so
encourage you to play with it thanks this is Lance from Lang chain I'm

going to talk about indexing uh and mulation indexing in particular for the
12th part of our rag from scratch series here so we previously talked about a few
different major areas we talk about query translation which takes a question and translates it in some way to
optimize for retrieval we talk about routing which is the process of taking a question routing it to the right data
source be it a vector store graph DB uh SQL DB we talked about queer
construction we dug into uh basically queer construction for Vector stores but of course there's also text SQL text to
Cipher um so now we're going to talk about indexing a bit in particular we're going to talk about indexing indexing
techniques for Vector Stores um and I want to highlight one particular method today called multi-representation
indexing so the high LEL idea here is derived a bit from a paper called
proposition indexing which kind of makes a simple observation you can think about decoupling raw documents and the unit you use for
retrieval so in the typical case you take a document you split it up in some
way to index it and then you embed the split directly
um this paper talks about actually taking a document splitting it in some
way but then using an llm to produce what they call a proposition which you can think of as like kind of a

distillation of that split so it's kind of like using an llm to modify that split in some way to distill it or make
it like a crisper uh like summary so to speak that's better optimized for
retrieval so that's kind of one highlight one piece of intuition so we actually taken that idea and we've kind
of built on it a bit in kind of a really nice way that I think is very well suited actually for long context llms so
the idea is pretty simple you take a document and you you actually distill it
or create a proposition like they show in the prior paper I kind of typically think of this as just produce a summary
of the document and you embed that summary so that summary is meant to be
optimized for retrieval so might contain a bunch of keywords from the document or like the big ideas such that when you embed the
summary you embed a question you do search you basically can find that document based upon this highly
optimized summary for retrieval so that's kind of represented here in your vector store but here's the catch you
independently store the raw document in a dock store and when you when you
basically retrieve the summary in the vector store you return the full document for the llm to perform
generation and this is a nice trick because at generation time now with long condex LMS for example the LM can handle
that entire document you don't need to worry about splitting it or anything you just simply use the summary to prod like
to create a really nice representation for fishing out that full dock use that full dock in generation there might be a
lot of reasons you want to do that you want to make sure the LM has the full context to actually answer the question so that's the big idea it's a nice trick
and let's walk through some code here we have a notebook all set up uh just like before we done some pip
installs um set to maybe I Keys here for lsmith um kind of here's a diagram now
let me show an example let's just load two different uh blog posts uh one is about agents one is about uh you know
human data quality um and what we're going to do is let's create a summary of
each of those so this is kind of the first step of that process where we're going from like the raw documents to summaries let's just have a look and
make sure those ran So Okay cool so the first DOC discusses you know building autonomous agents the second doc
contains the importance of high quality human data and training okay so that's pretty nice we have our summaries now
we're going to go through a process that's pretty simple first we Define a vector store that's going to index those
summaries now we're going to Define what we call like our our document storage is going to store the full documents okay
so this multiv Vector retriever kind of just pulls those two things together we basically add our Dock Store we had this
bite store is basically the the the full document store uh the vector store is our Vector store um and now this ID is

what we're going to use to reference between the chunks or the summaries and the full documents that's really it so
now for every document we'll Define a new Doc ID um and then we're basically going to like take our summary documents
um and we're going to extract um for each of our summaries we're going to get
the associated doc ID so we go um so let's go ahead and do that so we have
our summary docs which we add to the vector store we have our full documents uh our doc IDs and the full raw
documents which are added to our doc store and then let's just do a query Vector store like a similarity search on
our Vector store so memory and agents and we can see okay so we can extract you know from the summaries we can get
for example the summary that pertains to um a agents so that's a good thing now
let's go ahead and run a query get relevant documents on our retriever which basically combines the summaries
uh which we use for retrieval then the doc store which we use to get the full doc back so we're going to apply our
query we're going to basically run this and here's the key Point we've gotten
back the entire article um and we can actually if you
want to look at the whole thing we we can just go ahead and do this here we go so this is the entire article that we
get back from that search so it's a pretty nice trick again we query with just memory and agents um and we can
kind of go back to our diagram here we quered for memory and agents it started our summaries it found the summary
related to memory and agents it uses that doc ID to reference between the vector store and the doc store it fishes
out the right full doc returns us the full document in this case the full web page that's really it simple idea nice
way to go from basically like nice simple proposition style or summary
style indexing to full document retrieval which is very useful especially with long contact LMS thank

you hi this is Lance from Lang chain this is the 13th part of our rag from scratch series focused on a technique
called Raptor so Raptor sits within kind of an array of different indexing techniques
that can be applied on Vector Stores um we just talked about multi-representation indexing um we I
priv a link to a video that's very good talking about the different means of chunking so I encourage you to look at
that and we're going to talk today about a technique called Raptor which you can kind of think of it as a technique for
hierarchical indexing so the highle intuition is
this some questions require very detailed information from a corpus to
answer like pertain to a single document or single chunk so like we can call those low-level questions some questions require consolidation across kind broad swast of a document so across like many documents

or many chunks within a document and you can call those like higher level questions and so there's kind of this
challenge in retrieval and that typically we do like K nearest neighbors retrieval like we've been talking about
you're fishing out some number of chunks but what if you have a question that requires information across like five
six you know or a number of different chunks which may exceed you know the K
parameter in your retrieval so again when you typically do retrieval you might set a k parameter of three which
means you're retrieving three chunks from your vector store um and maybe you have a high very high level question
that could benefit from infation across more than three so this technique called raptor is basically a way to build a
hierarchical index of document summaries and the intuition is this you start with
a set of documents as your Leafs here on the left you cluster them and then you
Summarize each cluster so each cluster of similar documents um will consult
information from across your context which is you know your context could be a bunch of different splits or could
even be across a bunch of different documents you're basically capturing similar ones and you're consolidating
the information across them in a summary and here's the interesting thing you do that recursively until either you hit like a
limit or you end up with one single cluster that's a kind of very high level summary of all of your
documents and what the paper shows is that if you basically just collapse all
these and index them together as a big pool you end up with a really nice array of chunks that span the abstraction
hierarchy like you have a bunch of chunks from Individual documents that are just like more detailed chunks
pertaining to that you know single document but you also have chunks from these summaries or I would say like you
know maybe not chunks but in this case the summary is like a distillation so you know raw chunks on the left that
represent your leavs are kind of like the rawest form of information either raw chunks or raw documents and then you
have these higher level summaries which are all indexed together so if you have higher level questions they should
basically be more similar uh in sematic search for example to these higher level summary chunks if you have lower level
questions then they'll retrieve these more lower level chunks and so you have better semantic coverage across like the
abstraction hierarchy of question types that's the intuition they do a bunch of nice studies to show that this works
pretty well um I actually did a deep dive video just on this which I link
below um I did want to cover it briefly just at a very high level um so let's
actually just do kind of a code walkr and I've added it to this rack from scratch course notebook but I link over
to my deep dive video as well as the paper and the the full code notebook

which is already checked in is discussed at more length in the Deep dive the technique is a little bit detailed so I
only want to give you very high levels kind of overview here and you can look at the Deep dive video if you want to go
in more depth again we talked through this abstraction hierarchy um I applied this to a large set of Lang chain documents um so this is me loading basically all of our Lang chain expression language docs so this
is on the order of 30 documents you can see I do a histogram here of the token counts per document some are pretty big
most are fairly small less than you know 4,000 tokens um and what I did is I
indexed all of them um individually so the all those raw documents you can kind
of Imagine are here on the left and then I do um I do embedding I do clustering
summarization and I do that recursively um until I end up with in this case I
believe I only set like three levels of recursion and then I save them all my Vector store so that's like the highle
idea I'm applying this Raptor technique to a whole bunch of Lang chain documents
um that have fairly large number of tokens um so I do that um and yeah I use
actually use both CLA as well as open AI here um this talks through the
clustering method which they that they use which is pretty interesting you can kind of dig into that on your own if if you're really um interested this is a
lot of their code um which I cite accordingly um this is basically implementing the clustering method that
they use um and this is just simply the document embedding stage um this is like
basically embedding uh and clustering that's really it uh some text formatting
um summarizing of the clusters right here um and then this is just running
that whole process recursively that's really it um this is tree building so
basically I have the RO the rod docs let's just go back and look at Doc texts so this should be all my raw documents
uh so that's right you can see it here doc text is basically just the text in all those Lang chain documents that I
pulled um and so I run this process on them
right here uh so this is that recursive embedding cluster basically runs and produces is that tree here's the results
um this is me just going through the results and basically adding the result text to this list of uh texts um oh okay
so here's what I do this Leaf text is all the raw documents and I'm appending to that all the summaries that's all
it's going on and then I'm indexing them all together that's the key Point rag chain and there you have it that's
really all you do um so anyway I encourage you to look at this in depth it's a pretty interesting technique it
works well long with long contexts so for example one of the arguments I made is that it's kind of a nice approach to
consult information across like a span of large documents like in this particular case
my individual documents were lch expression language docs uh each each being somewhere in the order of you know
in this case like you know most of them are less than 4,000 tokens some pretty big but I index them all I cluster them

without any splits uh embed them cluster them build this tree um and go from
there and it all works because we now have llms that can go out to you know 100 or 200,000 up to million tokens and
Contex so you can actually just do this process for big swats of documents in place without any without any splitting
uh it's a pretty nice approach so I encourage you to think about it look at it watch the deep that video If you really want to go deeper on this um

thanks hi this is Lance from Lang chain this is the 14th part of our rag from scratch series we're going to I'm going
to be talking about an approach called cold bear um so we've talked about a few
different approaches for indexing and just as kind of a refresher indexing Falls uh kind of right down here in our
flow we started initially with career translation taking a question translating it in some way to optimize
retrieval we talked about routing it to a particular database we then talked about query construction so going from
natural language to the DSL or domain specific language for E any of the
databases that you want to work with those are you know metadata filters for Vector stores or Cipher
for graph DB or SQL for relational DB so that's kind of the flow we talked about today we talked about some indexing
approaches like multi-representation indexing we gave a small shout out to greet camer in the series on chunking uh
we talked about hierarchical indexing and I want to include one Advanced kind
embedding approach so we talked a lot about embeddings are obviously very Central to semantic similarity search um
and retrieval so one of the interesting points that's been brought up is that
embedding models of course take a document you can see here on the top and embed it basically compress it to a
vector so it's kind of a compression process you representing all the semantics of that document in a single
Vector you're doing the same to your question you're doing similarity search between the question embedding and the
document embedding um in order to perform retrieval you're typically taking the you know K most similar um
document abetting is given a question and that's really how you're doing it now a lot of people said well hey the
compressing a full document with all this Nuance to single Vector seems a little bit um overly restrictive right
and this is a fair question to ask um there's been some interesting approaches to try to address that and one is this
this this approach method called Co bear so the intuition is actually pretty
straightforward there's a bunch of good articles I link down here this is my little cartoon to explain it which I
think is hopefully kind of helpful but here's the main idea instead of just taking a document and compressing it
down to a single Vector basically single uh what we might call embedding Vector

we take the document we break it up into tokens so tokens are just like you know units of of content it depends on the
token areas you use we talked about this earlier so you basically tokenize it and you produce basically an embedding or
vector for every token and there's some kind of positional uh waiting that occurs when you do this process so you
obviously you look to look at the implementation understand the details but the intuition is that you're producing some kind of representation
for every token okay and you're doing the same thing for your question so
you're taking your question you're breaking into a tokens and you have some representation or vector per token
and then what you're doing is for every token in the question you're Computing the similarity across all the tokens in
the document and you're finding the max you're taking the max you're storing
that and you're doing that process for all the tokens in the question so again
token two you compare it to every token in the in the document compute the
Max and then the final score is in this case the sum of the max similarities uh
between every question token and any document token so it's an interesting
approach uh it reports very strong performance latency is definitely a
question um so kind of production Readiness is something you should look into but it's a it's an approach that's
worth mentioning here uh because it's pretty interesting um and let's walk through
the code so there's actually nice Library called
rouille which makes it very easy to play with Co bear um she's pip install it here I've already done that and we can
use one of their pre-train models to mediate this process so I'm basically following their documentation this is kind of what they recommended um so I'm
running this now hopefully this runs somewhat quickly I'm not sure I I previously have loaded this model so hopefully it won't take too long and yeah you can see it's pretty quick uh I'm on a Mac M2 with 32
gigs um so just as like a context in terms of my my system um this is from their documentation we're just grabbing
a Wikipedia page this is getting a full document on Miyazaki so that's cool
we're going to grab that now this is just from their docs this is basically how we create an index so we provide the
you know some index name the collection um the max document length and yeah you should look at their documentation for
these flags these are just the defaults so I'm going to create my index um so I get some logging here so it it's working
under the hood um and by the way I actually have their documentation open so you can kind of follow along um
so um let's see yeah right about here so you can kind of follow this indexing
process to create an index you need to load a train uh a trained model this can be either your own pre-train model or
one of ours from The Hub um and this is kind of the process we're doing right now create index is just a few lines of
code and this is exactly what we're doing um so this is the you know my documents and this is the indexing step

that we just we just kind of walk through and it looks like it's done um so you get a bunch of logging here
that's fine um now let's actually see if this works so we're going to run drag search what an emotion Studio did Miaki
found set our K parameter and we get some results okay so it's running and
cool we get some documents out so you know it seems to work now what's nice is you can run this within lighting chain
as a liting chain retriever so that basically wraps this as a lighting chain Retriever and then you can use it freely
as a retriever within Lang chain it works with all the other different LMS and all the other components like rankers and so forth that we talk
through so you can use this directly as a retriever let's try this out and boom
nice and fast um and we get our documents again this is a super simple test example you should run this maybe
on more complex cases but it's pretty pretty easy spin up it's a really interesting alternative indexing
approach um using again like we talked through um a very different algorithm
for computing do similarity that may work better I think an interesting regime to consider this would be longer
documents so if you want like longer um yeah if if you basically want kind of
long context embedding I think you should look into for example the uh Max
token limits for this approach because it partitions the document into into each token um I would be curious to dig
into kind of what the overall context limits are for this approach of coar but it's really interesting to consider and
it reports very strong performance so again I encourage you to play with it and this is just kind of an intro to how to get set up and to start experimenting
with it really quickly thanks hi this is Lance from Lang chain

I'm going to be talking about using langra to build a diverse and sophisticated rag
flows so just to set the stage the basic rag flow you can see here starts with a
question retrieval of relevant documents from an index which are passed into the
context window of an llm for generation of an answer ground in your documents that's kind of the basic
outline and we can see it's like a very linear path um in practice though you often
encounter a few different types of questions like when do we actually want to retrieve based upon the context of the
question um are the retrieve documents actually good or not and if they're not good should we discard them and then how
do we loot back and retry retrieval with for example and improved question so these types of questions
motivate an idea of active rag which is a process where an llm actually decides
when and where to retrieve based upon like existing retrievals or existing
Generations now when you think about this there's a few different levels of control that you have over an llm in a
rag application the base case like we saw with our chain is just use an llm to
choose a single steps output so for example in traditional rag you feed it
documents and it decides to generation so it's just kind of one step now a lot

of rag workflows will use the idea of routing so like given a question should I route it to a vector store or a graph
DB um and we have seen this quite a bit now this newer idea that I want to
introduce is how do we build more sophisticated logical
flows um in a rag pipeline um that you let the llm choose
between different steps but specify all the transitions that are
available and this is known as we call a state machine now there's a few different
architectures that have emerged uh to build different types of rag chains and of course chains are
traditionally used just for like very basic rag but this notion of State machine is a bit newer and Lang graph
which we recently released provides a really nice way to build State machines for Rag and for other
things and the general idea here is that you can lay out more diverse and
complicated rag flows and then Implement them as graphs
and it kind of motivates this more broad idea of of like flow engineering and thinking through the actual like
workflow that you want and then implementing it um and we're gonna actually do that right now so I'm GNA Pi
a recent paper called CAG corrective rag which is really a nice method um for
active rag that incorporates a few different ideas um so first you retrieve documents
and then you grade them now if at least one document
exceeds the threshold for relevance you go to generation you
generate your answer um and it does this knowledge
refinement stage after that but let's not worry about that for right now it's
kind of not essential for understanding the basic flow here so again you do a
grade for relevance for every document if any is relevant you
generate now if they're all ambiguous or incorrect based upon your
grader you retrieve from an external Source they use web
search and then they pass that as their context for answer
generation so it's a really neat workflow where you're doing retrieval just like with basic rag but then you're
reasoning about the documents if they're relevant go ahead and at least one is relevant go ahead and generate if
they're not retrieve from alternative source and then pack that into the context and generate your
answer so let's see how we would implement this as a estate machine using
Lang graph um we'll make a few simplifications
um we're going to first decide if any documents are relevant we'll go ahead
and do the the web search um to supplement the output so that's
just like kind of one minor modification um we'll use tab search for web search um we use Query writing to
optimize the search for uh to optimize the web search but it follows a lot of the the intuitions of the main paper uh
small note here we set the Tav API key and another small mode I've already set
my lsmith API key um with which we'll see is useful a bit later for observing
the resulting traces now I'm going to index three blog

posts that I like um I'm going to use chroma DB I'm G use open ey embeddings I'm going to run
this right now this will create a vector store for me from these three blog
posts and then what I'm going to do is Define
State now this is kind of the core object that going to be passed around my graph that I'm going to
modify and right here is where I Define it and the key point to note right now is it's just a dictionary and it can
contain things that are relevant for rag like question documents generation and
we'll see how we update that in in in a little bit but the first thing to note is we Define our state and this is
what's going to be modified in every Noe of our graph now here's really the Crux of it
and this is the thing I want to zoom in on a little bit um
so when you kind of move from just thinking about promps to thinking about overall flows it it's like kind of a fun
and interesting exercise I kind of think about this as it's been mentioned on Twitter a little bit more like flow
engineering so let's think through what was actually done in the paper and what
modifications to our state are going to happen in each stage so we start with a
question you can see that on the far left and this kind of state is represent as a dictionary like we have we start
with a question we perform retrieval from our Vector store which we just created that's going to give us
documents so that's one node we made an an adjustment to our state by adding
documents that's step one now we have a second node where we're going to grade the documents and
in this node we might filter some out so we are making a modification to state which is why it's a node so we're going
to have a greater then we're going to have what we're going to call a conditional Edge
so we saw we went from question to retrieval retrieval always goes to grading and now we have a
decision if any document is irrelevant we're going to go ahead and
do web search to supplement and if they're all relevant will go to generation it's a minor kind of a minor kind of logical uh decision ision that we're going to
make um if any are not relevant we'll transform the query and we'll do web
search and we'll use that for Generation so that's really it and that's how we can kind of think about our flow and how
our States can be modified throughout this flow now all we then need to do and I I
kind of found spending 10 minutes thinking carefully through your flow
engineering is really valuable because from here it's really implementation
details um and it's pretty easy as you'll see so basically I'm going to run
this code block but then we can like walk through some of it I won't show you everything so it'll get a little bit boring but really all we're doing is
we're finding functions for every node that take in the state and modify in
some way that's all it's going on so think about retrieval we run retrieval we take in state remember it's a dict we
get our state dick like this we extract one keyy question from our dick we pass that to a retriever we get

documents and we write back out State now with documents key added that's
all generate going to be similar we take in state now we have our question and
documents we pull in a prompt we Define an llm we do minor post processing on
documents we set up a chain for retrieval uh or sorry for Generation which is just going to be take our
prompt pump Plum that to an llm partially output a string and we run it
right here invoking our documents in our question to get our answer we write that
back to State that's it and you can kind of follow here for every node we just Define a function
that performs the state modification that we want to do on that node grading documents is going to be
the same um in this case I do a little thing extra here because I actually
Define a identic data model for my grader so that the output of that
particular grading chain is a binary yes or no you can look at the code make sure
it's all shared um and that just makes sure that our output is is very deterministic so that we then can down
here perform logical filtering so what you can see here is um we Define this
search value no and we iterate through our documents we grade them if any
document uh is graded as not relevant we flag this search thing to yes that means
we're going to perform web search we then add that to our state dict at the end so run web search now that value is
true that's it and you can kind of see we go through some other nodes here there's web search
node um now here is where our one conditional Edge we Define right here
this is where where we decide to generate or not based on that search key so we again get our state let's extract
the various values so we have this search value now if search is yes we
return the next no that we want to go to so in this case it'll be transform query
which will then go to web search else we go to generate so what we can see is we laid
out our graph which you can kind of see up here and now we Define functions for all
those nodes as well as the conditional Edge and now we scroll down all we have
to do is just lay that out here again as our flow and this is kind of what you might think of as like kind of flow
engineering where you're just laying out the graph as you drew it where we have
set our entry point as retrieve we're adding an edge between retrieve and grade documents so we went retrieval
grade documents we add our conditional Edge depending on the grade either transform the query go to web search or
just go to generate we create an edge between transform the query and web search then web search to generate and
then we also have an edge generate to end and that's our whole graph that's it so we can just run
this and now I'm going to ask a question so let's just say um how does agent
memory work for example let's just try that and what this is going to do is going to print out what's going on as we
run through this graph so um first we going to see output from retrieve this is going to be all of our

documents that we retrieved so that's that's fine this just from our our retriever then you can see that we're
doing a relevance check across our documents and this is kind of interesting right you can see we grading
them here one is grade as not relevant um and okay you can see the
documents are now filtered because we removed the one that's not relevant and because one is not relevant we decide
okay we're going to just transform the query and run web search and um you can see after query
transformation we rewrite the question slightly we then run web search um and you can see from web
search it searched from some additional sources um which you can actually see
here it's appended as a so here it is so here it's a new document appended from web search
which is from memory knowledge requirements so it it basically looked up some AI architecture related to
memory uh web results so that's fine that's exactly what we want to do and then um we generate a
response so that's great and this is just showing you everything in kind of gory detail but I'm going to show you
one other thing that's that's really nice about this if I go to lsmith
I have my AP I ke set so all my Generations are just logged to to lsmith
and I can see my Lang graph run here now what's really cool is this shows me all
of my nodes so remember we had retrieve
grade we evaluated the grade because one was irrelevant we then went ahead and
transformed the query we did a web search we pended that to our context you can see all those steps are laid out
here in fact you can even look at every single uh grader and its output I will
move this up slightly um so you can see the the different scores for grades okay so this
particular retrieval was graded as as not relevant so that's fine that that can happen in some cases and because of
that um we did a query transformation so we modified the question slightly how
does memory how does the memory system an artificial agents function so it's just a minor rephrasing of the question
we did this Tav web search this is where it queried from this particular blog
post from medium so it's like a sing web query we can like sanity check it and then what's need is we can go to our
generate step look at open Ai and here's our full prompt how does the memory system in our official agents function
and then here's all of our documents so this is the this is the web search as well as we still have the Rel chunks
that were retrieved from our blog posts um and then here's our answer so that's
really it you can see how um really moving from the notion of just like I'll
actually go back to the original um moving
from uh I will try to open this up a little bit um
yeah I can see my face still um the transition from laying out
simple chains to flows is a really interesting and
helpful way of thinking about why graphs are really interesting because you can encode more sophisticated logical

reasoning workflows but in a very like clean and well-engineered way
where you can specify all the transitions that you actually want to have executed um and I actually find this way
of thinking and building kind of logical uh like workflows really
intuitive um we have a blog post coming out uh tomorrow that discusses both
implementing self rag as well as C rag for two different active rag approaches using using uh this idea of of State
machines and Lang graph um so I encourage you to play with it uh I found
it really uh intuitive to work with um I also found uh inspection of traces to be
quite intuitive using Lang graph because every node is enumerated pretty clearly
for you which is not always the case when you're using other types of of more complex reasoning approaches for example
like agents so in any case um I hope this was helpful and I definitely
encourage you to check out um kind of this notion of like flow engineering using Lang graph and in the context of
rag it can be really powerful hopefully as you've seen here thank

you hey this is Lance from Lang chain I want to talk to a recent paper that I saw called adaptive rag which brings
together some interesting ideas that have kind of been covered in other videos but this actually ties them all together in kind of a fun way so the the
two big ideas to talk about here are one of query analysis so we've actually done
kind of a whole rag from scratch series that walks through each of these things in detail but this is a very nice
example of how this comes together um with some other ideas we've been talking about so query analysis is typically the
process of taking an input question and modifying in some way uh to better
optimize retrieval there's a bunch of different methods for this it could be decomposing it into sub questions it
could be using some clever techniques like stepb back prompting um but that's
kind of like the first stage of query analysis then typically you can do routing so you route a question to one
of multiple potential sources it could be one or two different Vector stores it could be relational DB versus Vector
store it could be web search it could just be like an llm fallback right so
this is like one kind of big idea query analysis right it's kind of like the front end of your rag pipeline it's
taking your question it's modifying it in some way it's sending it to the right place be it a web search be it a vector
store be it a relational DB so that's kind of topic one now topic two is
something that's been brought up in a few other videos um of what I kind of call Flow engineering or adaptive rag
which is the idea of doing tests in your rag pipeline or in your rag inference flow uh to do things like check
relevance documents um check whether or not the answer contains hallucinations
so this recent blog post from Hamil Hussein actually covers evaluation in in some really nice detail and one of the

things he highlighted explicitly is actually this topic so he talks about unit tests and in particular
he says something really interesting here he says you know unlike typical unit tests you want to organize these assertions in places Beyond typical unit
testing such as data cleaning and here's the key Point automatic retries during
model inference that's the key thing I want to like draw your attention to to it's a really nice approach we've talked
about some other papers that do that like corrective rag self rag but it's also cool to see it here and kind of
encapsulated in this way the main idea is that you're using kind of unit tests in your flow to make Corrections like if
your retrieval is bad you can correct from that if your generation has hallucinations you can correct from that
so I'm going to kind of draw out like a cartoon diagram of what we're going to do here and you can kind of see it here
we're starting with a question we talked about query analysis we're going to take our question and we're going to decide
where it needs to go and for this particular toy example I'm going to say either send it to a vector store send it
to web search or just have the llm answer it right so that's like kind of my fallback Behavior then we're going to bring in
that idea of kind of online flow engineering or unit testing where I'm
going to have my retrieval either from the VOR store or web search I'm then going to ask is this actually relevant
to the question if it isn't I'm actually going to kick back to web sech so this is a little bit more relevant in the
case if I've routed to to the vector store done retrieval documents aren't relevant I'll have a fallback
mechanism um then I'm going to generate I check for hallucinations in my generation and then I check for um for
whether or not the the generation actually answers the question then I return my answer so again we're tying together two ideas one is query analysis
like basically taking a question routing it to the right place modifying it as needed and then kind of online unit
testing and iterative flow feedback so to do this I've actually
heard a lot of people talk online about command r a new model release from gooh here it has some pretty nice properties
that I was kind of reading about recently so one it has nice support for Tool use and it does support query
writing in the context of tool use uh so this all rolls up in really nice
capabilities for routing it's kind of one now two it's small it's 35 billion
parameter uh it's actually open weight so you can actually run this locally and I've tried that we can we can talk about
that later uh so and it's also fast served via the API so it's kind of a small model and it's well tuned for rag
so I heard a lot of people talking about using coher for Rag and it has a large context 120,000 tokens so this like a

nice combination of properties it supports to and routing it's small and fast so it's like quick for grading and
it's well tuned for rag so it's actually a really nice fit for this particular workflow where I want to do query
analysis and routing and I want to do kind of online checking uh and rag so
kind of there you go now let's just get to the coding bit so I have a notebook kind of like usual I've done a few pip
installs you can see it's nothing exotic I'm bringing Lang chain coh here I set my coher API key now I'm just going to
set this Lang chain project within lsmith so all my traces for this go to that project and I have enabled tracing
so I'm using Langs Smith here so we're going to walk through this flow and let's do the first thing let's just
build a vector store so I'm going to build a vector store using coherent beddings with chroma open source Vector
DB runs locally from three different web pages on blog post that I like so it pertains to agents prompt engineering
and adversarial attacks so now I have a retriever I can run retriever invoke and
I can ask a question about you know agent memory agent
memory and there we go so we get documents back so there we go we have a retriever now now here's where I'm going
to bring in coh here I also want a router so you look at our flow the first step is this routing stage right so what
I'm going to do is I'm guess we going to find two tools a web search tool and a
vector store tool okay in my Preamble is just going to say you're an expert routing user questions to Vector store
web search now here's the key I tell it what the vector store has so again my
index my Vector store has agents prompt engineering adial tax I just repeat that here agents prompt adversarial tax so it
knows what's in the vector store um so use it for questions on these topics otherwise use web search so that's it I
use command R here now I'm going to bind these tools to the model and attach the Preamble and I have a structured LM
router so let's give it a let's give this a few tests just to like kind of sandbox this a little bit
so I can inval here's my chain I have a router prompt I pass that to the structured LM router which I defined
right here and um let's ask a few different questions like who will the Bears draft in the NFL draft with types
of agent memory and Hi how are you so I'm going to kick that off and you can
see you know it does web search it does it goes to Vector store and then actually returns this false so that's
kind of interesting um this is actually just saying if it does not use either tool so
for that particular query web search or the vector store was inappropriate it'll just say hey I didn't call one of those
tools so that's interesting we'll use that later so that's my router tool now
the second thing is my grader and here's where I want to show you something really nice that is generally useful uh

for many different problems you might encounter so here's what I'm doing I'm defining a data model uh for My Grade so
basically grade documents it's going to have this is a pantic object it is just
basically a binary score here um field specified here uh documents are relevant
to the question yes no I have a preamble your grer assessing relevance of retrieve documents to a user question um
blah blah blah so you know and then basically give it a b score yes no I'm using command R but here's the catch I'm
using this wi structured outputs thing and I'm passing my grade documents uh data model to that that so this is the
key thing we can test this right now as well it's going to return an object
based on the schema I give it which is extremely useful for all sorts of use cases and let's actually Zoom back up so
we're actually right here so this greater stage right I want to constrain
the output to yes no I don't want any preambles I want anything because the logic I'm going to build in this graph
is going to require a yes no binary response from this particular Edge in our graph
so that's why this greater tool is really useful and I'm asking like a mock
question types of agent memory I do a retriever I do a retrieval from our Vector store I get the tuck and I test
it um I invoke our greater retrieval grater chain with the question the doc
text and it's relevant as we would expect so that's good but again let's
just kind of look at that a little bit more closely what's actually happening under the hood here here's the pantic object we passed
here's the document in question I'm providing basically it's converting this object into coher function schema it's
binding that to the llm we pass in the document question it returns an object basic a Json string
per our pantic schema that's it and then it's just going to like parse that into
a pantic object which we get at the end of the day so that's what's happening under the hood with this with structured output thing but it's extremely useful
and you'll see we're going to use that a few different places um um because we want to ensure that in our in our flow
here we have three different grading steps and each time we want to constrain the output to yes no we're going to use
that structured output more than once um this is just my generation so this is good Old Rag let's just make
sure that works um I'm using rag chain typical rag prompt again I'm using
cohere for rag pretty easy and yeah so the rag piece works that's totally fine
nothing to it crazy there um I'm going to find this llm fallback so this is
basically if you saw a router chain if it doesn't use a tool I want to fall
back and just fall back to the llm so I'm going to kind of build that as a little chain here so okay this is just a
fallback I have my Preamble just you're you're an assistant answer the question based upon your internal knowledge so
again that fallback behavior is what we have here so what we've done already is
we defined our router piece we've defined our our basic retrieval our Vector store we already have here um

we've defined our first logic or like grade check and we defined our fallback
and we're just kind of roll through the parts of our graph and Define each piece um so I'm going to have two other
graders and they're going to use the same thing we just talked about slightly different data model I mean same output
but actually just slightly different uh prompt um and you know descript destion
this in this case is the aners grounded the facts yes no this is my hallucination grater uh and then I have an answer
grader as well and I've also run a test on each one and you can see I'm getting binary this this these objects out have
a binary score so this a pantic object with a binary score uh and that's exactly what we want cool
and I have a Search tool so that's really nice we've actually gone through and we've kind of laid out I have like a
router I've tested it we have a vector story tested we've tested each of our graders here we've also tested
generation of just doing rag so we have a bunch of pieces built here we have a fallback piece we have web search now
the question is how do I Stitch these together into this kind of flow and for that I I like to use Lang graph we'll
talk a little about Lang graph versus agents a bit later but I want to show you why this is really easy to do using
Lang graph so what's kind of nice is I've kind of laid out all my logic here we've tested individually and now all
I'm going to do is I'm going to first lay out uh the parts of my graph so what you're going
to notice here is first there's a graph state so this state represents kind of the key parts of the graph or the key
kind of objects within the graph that we're going to be modifying so this is basically a graph centered around rag
we're going to have question generation and documents that's really kind of the main things we're going to be working with in our
graph so then you're going to see something that's pretty intuitive I think what you're going to see is we're
going to basically walk through this flow and for each of these little circles we're just going to find a function and these uh little squares or
these these you can think about every Circle as a node and every kind of diamond here as as an edge or
conditional Edge so that's actually what we're going to do right now we're going to lay out all of our nodes and edges
and each one of them are just going to be a function and you're going to see how we do that right now so I'm going to
go down here I def find my graph state so this is what's going to be kind of modified and propagated throughout my graph now all I'm going to do is I'm
just going to find a function uh for each of those nodes so let me kind of go side by side and show you the diagram
and then like kind of show the nodes next to it so here's the diagram so we have uh a retrieve node so

that kind of represents our Vector store we have a fallback node that's this piece we have a generate node so that's
basically going to do our rag you can see there we have a grade documents node
kind of right here um and we have a web search node so that's right here cool now here's where
we're actually to find the edges so you can see our edges are the pieces of the graph that are kind of making different decisions so this route question Edge
basic conditional Edge is basically going to take an input question and decide where it needs to go and that's
all we're doing down here it kind of follows what we did up at the top where we tested this individually so recall we
basically just invoke that question router returns our source now remember if tool calls were not in the source we
do our fall back so we show actually showed that all the way up here remember this if tool calls is not in the
response this thing will just be false so that means we didn't either we didn't call web search and we didn't call uh
our retriever tool so then we're just going to fall back
um yep right here and this is just like uh you know a catch just in case a tool
could make a decision but most interestingly here's where we choose a data source basically so um this is the
output of our tool call we're just going to fish out the name of the tool so
that's data source and then here we go if the data source is web search I'm returning web search as basically the
next node to go to um otherwise if it's Vector store we return Vector store as the next node to go to so what's this
search thing well remember we right up here Define this node web search that's
it we're just going to go to that node um what's this Vector store um you'll
see below how we can kind of tie these strings that we returned from the conditional Edge to the node we want to
go to that's really it um same kind of thing here decide to generate that's
going to roll in these two conditional edges into one um and basically it's going to do if there's no documents so
basic basically if we filtered out all of our documents from this first test here um then what we're going to do is
we've decided all documents are not relevant to the question and we're going to kick back to web search exactly as we
show here so that's this piece um otherwise we're going to go to generate
so that's this piece so again in these conditional edges you're basically implementing the logic that you see in
our diagram right here that's all that's going on um and again this is just
implementing the final two checks uh for hallucinations and and answer relevance um
and um yep so here's our hallucination grader we then extract the grade if the
if basically there are hallucinations um oh sorry in this case the grade actually yes means that the
answer is grounded so we say answer is actually grounded and then we go to the next step we go to the next test that's

all this is doing it's just basically wrapping this logic that we're implementing here in our graph so that's
all that's going on and let's go ahead and Define all those things so nice we have all that um now we can actually go
down a little bit and we can pull um this is actually where we stitch
together everything so all it's happening here is you see we defined all these functions up here we just add them
as nodes in our graph here and then we build our graph here basically by by
basically laying out the flow or the connectivity between our nodes and edges so you know you can look at this
notebook to kind of study in a bit of detail what's going on but frankly what I like to do here typically just draw
out a graph kind of like we did up here and then Implement uh the Lo
logical flow here in your graph as nodes and edges just like we're doing here
that's all that's happening uh so again we have like our entry point is the router
um this is like the output is this is basically directing like here's what the router is outputting and here's the next
node to go to so that's it um and then for each node we're kind of applying
like we're saying like what's what's the flow so web search goes to generate after um and retrieve goes to grade
documents grade documents um kind of is is like is a conditional Edge um
depending on the results we either do web search or generate and then our second one we go from generate to uh
basically this grade uh generation versus documents in question based on the output of that we either have
hallucinations we regenerate uh we found that the answer is not useful we kick back to web search or we end um finally
we have that llm fallback and that's also if we go to the fallback we end so what you're seeing here is actually the
the logic flow we're laying out in this graph matches the diagram
that we laid out up top I'm just going to copy these over and I'll actually go then back to the diagram and and kind of
underscore that a little bit more so here is the flow we've laid out again
here is our diagram and you can kind of look at them side by side and see how they basically match up so here's kind
of our flow diagram going from basically query analysis that's this thing this
route question and you can see web search Vector store LM fallback LM
fallback web search vector store so those are like the three options that can come out of this conditional Edge
and then here's where we connect so if we go to web search then basically we next go to generate so that's kind of this whole flow um now if we go to
retrieve um then we're going to grade so that's it um and you know it follows kind of as
you can see here that's really it uh so it's just nice to draw the these
diagrams out first and then it's pretty quick to implement each node and each Edge just as a function and then stitch
them together in a graph just like I show here and of course we'll make sure this code's publ
so you can use it as a reference um so there we go now let's
try a few a few different test questions so like what player the Bears to draft

and NFL draft right let's have a look at that and they should print everything it's doing as we go so okay this is
important route question it just decides to route to web search that's good it doesn't go to our Vector store this is a
current event not related to our Vector store at all it goes to web search um and then it goes to generate so that's
what we'd expect so basically web search goes through to generate um and we check hallucinations
Generations ground the documents we check generation versus question the generation addresses the question the
Chicago Bears expected to draft Caleb Williams that's right that's that's the consensus so cool that works now let's
ask a question related to our Vector store what are the types of agent memory we'll kick this off so we're routing
okay we're routing to rag now look how fast this is that's really fast so we basically whip through that document
grading determine they're all relevant uh we go to decision to generate um we check hallucinations we
check answer versus question and there are several types of memory stored in the human brain memory can also be
stored in G of Agents you have LM agents memory stream retrieval model and and reflection mechanism so it's
representing what's captured on the blog post pretty reasonably now let me show you something else is kind of nice I can go to Langs Smith and I can go to my
projects we create this new project coher adaptive rag at the start and everything is actually logged there
everything we just did so I can open this up and I can actually just kind of look through all the stages of my Lang
graph to here's my retrieval stage um here's my grade document stage and we
can kind of audit the grading itself we kind of looked at this one by one previously but it's actually pretty nice
we can actually audit every single individual document grade to see what's happening um we can basically go through
um to this is going to be one of the other graders here
um yep so this is actually going to be the hallucination grading right here uh
and then this is going to be the answer grading right here so that's really it you can kind of walk through the entire graph you can you can kind of study
what's going on um which is actually very useful so it looks like this worked pretty well um and finally let's just
ask a question that should go to that fallback uh path down at the bottom like not related at all to our Vector store
current events and yeah hello I'm doing well so it's pretty neat we've seen in maybe 15 minutes we've from scratch
built basically a semi- sophisticated rag system that has agentic properties we've done in Lang graph we've done with
coher uh command R you can see it's pretty darn fast in fact we can go to Langs Smith and look at so this whole
thing took 7 seconds uh that is not bad let's look at the most recent one so

this takes one second so the fallback mechanism to the LM is like 1 second um
the let's just look here so 6 seconds for the initial uh land graph so this is
not bad at all it's quite fast it done it does quite a few different checks we do routing uh and then we have kind of a
bunch of nice fallback behavior and inline checking uh for both relevance hallucinations and and answer uh kind of
groundedness or answer usefulness so you know this is pretty nice I definitely
encourage you to play with a notebook command R is a really nice option for this due to the fact that is tool use routing uh small and quite fast and it's
really good for Rags it's a very nice kind of uh a very nice option for workflows like this and I think you're
going to see more and more of this kind of like uh adaptive or self-reflective rag um just because this is something
that a lot systems can benefit from like a a lot of production rack systems kind of don't necessarily have
fallbacks uh depending on for example like um you know if the documents
retrieved are not relevant uh if the answer contains hallucinations and so forth so this opportunity to apply
inline checking along with rag is like a really nice theme I think we're going to see more and more of especially as model
inference gets faster and faster and these checks get cheaper and cheaper to do kind of in the inference Loop
now as a final thing I do want to bring up the a point about you know we've shown this Lang graph stuff what about
agents you know how do you think about agents versus Lang graph right and and I think the way I like to frame this is
that um Lang graph is really good for um flows that you have kind of very
clearly defined that don't have like kind of open-endedness but like in this case we know the steps we want to take
every time we want to do um basically query analysis routing and then we want to do a three grading steps and that's
it um Lang graph is really good for building very reliable flows uh it's
kind of like putting an agent on guard rails and it's really nice uh it's less
flexible but highly reliable and so you can actually use smaller faster models with langra so that's the thing I like
about we saw here command R 35 billion parameter model works really well with langra quite quick we' were able to
implement a pretty sophisticated rag flow really quickly 15 minutes um in
time is on the order of like less than you know around 5 to 6 seconds so so pretty good right now what about agents
right so I think Agents come into play when you want more flexible workflows you don't want to necessarily follow a
defined pattern a priori you want an agent to be able to kind of reason and make of open-end decisions which is
interesting for certain like long Horizon planning problems you know agents are really interesting the catch is that
reliability is a bit worse with agents and so you know that's a big question a lot of people bring up and that's kind

of where larger LMS kind of come into play with a you know there's been a lot of questions about using small LMS even
open source models with agents and reliabilities kind of continuously being an issue whereas I've been able to run
these types of land graphs with um with uh like mraw or you know command R
actually is open weights you can run it locally um I've been able to run them very reproducibly with open source models on my laptop um so you know I
think there's a tradeoff and Comm actually there's a new coher model coming out uh believe command R plus
which uh is a larger model so it's probably more suitable for kind of more open-ended agentic use cases and there's
actually a new integration with Lang chain that support uh coher agents um which is quite nice so I think it's it's
worth experimenting for certain problems in workflows you may need more open-ended reasoning in which case use an agent with a larger model otherwise
you can use like Lang graph for more uh a more reliable potential
but con strain flow and it can also use smaller models faster LMS so those are some of the trade-offs to keep in mind
but anyway encourage you play with a notebook explore for yourself I think command R is a really nice model um I've
also been experimenting with running it locally with AMA uh currently the quantise model is like uh two bit
quantise is like 13 billion uh or so uh yeah 13 gigs it's it's a little bit too
large to run quickly locally for me um inference for things like rag we're
on the order of 30 seconds so again it's not great for a live demo but it does work it is available on a llama so I
encourage you to play with that I have a Mac M2 32 gig um so you know if I if you're a larger machine then it
absolutely could be worth working with locally so encourage you to play with that anyway hopefully this was useful
and interesting I think this is a cool paper cool flow um coher command R is a nice option for these types of like
routing uh it's quick good with Lang graph good for rag good for Tool use so
you know have a have a look and uh you know reply anything uh any feedback in the comments

thanks hi this is Lance from Lang chain this is a talk I gave at two recent meetups in San Francisco called is rag
really dead um and I figured since you know a lot of people actually weren't able to make those meetups uh I just
record this and put this on YouTube and see if this is of interest to folks um
so we all kind of recognize that Contex windows are getting larger for llms so on the x-axis you can see the tokens
used in pre-training that's of course you know getting larger as well um proprietary models are somewhere over
the two trillion token regime we don't quite know where they sit uh and we've all the way down to smaller models like
52 trained on far fewer tokens um but what's really notable is on the y axis

you can see about a year ago da the art models were on the order of 4,000 to
8,000 tokens and that's you know dozens of pages um we saw Claude 2 come out
with the 200,000 token model earlier I think it was last year um gbd4 128,000
tokens now that's hundreds of pages and now we're seeing Claud 3 and Gemini come out with million token models so this is
hundreds to thousands of pages so because of this phenomenon people have been kind of wondering is rag dead if
you can stuff you know many thousands of pages into the context window llm why do
you need a reteval system um it's a good question spoke sparked a lot of interesting debate on Twitter um and
it's maybe first just kind of grounding on what is rag so rag is really the process of reasoning and retrieval over
chunks of of information that have been retrieved um it's starting with you know documents that are indexed um they're
retrievable through some mechanism typically some kind of semantic similarity search or keyword search other mechanisms
retriev docs should then pass to an llm and the llm reasons about them to ground response to the question in the retrieve
document so that's kind of the overall flow but the important point to make is that typically it's multiple documents
and involve some form of reasoning so one of the questions I asked recently is you know if long
condex llms can replace rag it should be able to perform you know multia retrieval and reasoning from its own
context really effectively so I teamed up with Greg Cameron uh to kind of pressure test this and he had done some
really nice needle the Haack analyses already focused on kind of single facts
called needles placed in a Hy stack of Paul Graham essays um so I kind of
extended that to kind of mirror the rag use case or kind of the rag context uh
where I took multiple facts so I call it multi needle um I buil on a funny needle
in the HTO challenge published by anthropic where they add they basically placed Pizza ingredients in the context
uh and asked the LM to retrieve this combination of pizza ingredients I did I kind of Rift on that and I basically
split the pizza ingredients up into three different needles and place those three ingredients in different places in
the context and then ask the um to recover those three ingredients um from
the context so again the setup is the question is what the secret ingredients need to build a perfect Pizza the
needles are the ingredients figs Pudo goat cheese um I place them in the
context at some specified intervals the way this test works is you can basically set the percent of context you want to
place the first needle and the remaining two are placed at roughly equal intervals in the remaining context after
the first so that's kind of the way the test is set up now it's all open source by the way the link is below so needs
are placed um you ask a question you promp L them with with kind of um with

this context and the question and then produces the answer and now the the framework will grade the response
both one are you know all are all the the specified ingredients present in the
answer and two if not which ones are missing so I ran a bunch of analysis on
this with GPD 4 and came kind of came up with some with some fun results um so you can see on the left here what this
is looking at is different numbers of needles placed in 120,000 token context
window for gbd4 and I'm asking um gbd4 to retrieve
either one three or 10 needles now I'm also asking it to do reasoning on those
needles that's what you can see in those red bars so green is just retrieve the ingredients red is reasoning and the
reasoning challenge here is just return the first letter of each ingredient so we find is basically two things the
performance or the percentage of needles retrieved drops with respect to the number of needles that's kind of
intuitive you place more facts performance gets worse but also it gets worse if you ask it to reason so if you
say um just return the needles it does a little bit better than if you say return
the needles and tell me the first letter so you overlay reasoning so this is the first observation more facts is harder
uh and reasoning is harder uh than just retrieval now the second question we ask is where are these needles actually
present in the context that we're missing right so we know for example um
retrieval of um 10 needles is around 60% so where are the missing needles in the
context so on the right you can see results telling us actually which specific needles uh are are the model
fails to retrieve so what we can see is as you go from a th000 tokens up to
120,000 tokens on the X here and you look at needle one place at the start of
the document to needle 10 placed at the end at a th000 token context link you
can retrieve them all so again kind of match what we see over here small well
actually sorry over here everything I'm looking at is 120,000 tokens so that's really not the point uh the point is
actually smaller context uh better retrieval so that's kind of point one um
as I increase the context window I actually see that uh there is increased
failure to retrieve needles which you see can see in red here towards the start of the
document um and so this is an interesting result um and it actually matches what Greg saw with single needle
case as well so the way to think about it is it appears that um you know if you
for example read a book and I asked you a question about the first chapter you might have forgotten it same kind of
phenomenon appears to happen here with retrieval where needles towards the start of the context are are kind of
Forgotten or are not well retrieved relative to those of the end so this is an effect we see with gbd4 it's been
reproduced quite a bit so ran nine different trials here Greg's also seen this repeatedly with single needle so it
seems like a pretty consistent result and there's an interesting point I put this on Twitter and a number of

folks um you know replied and someone sent me this paper which is pretty interesting and it mentions recency bias
is one possible reason so the most informative tokens for predicting the next token uh you know are are are
present close to or recent to kind of where you're doing your generation and so there's a bias to attend to recent
tokens which is obviously not great for the retrieval problem as we saw here so
again the results show us that um reasoning is a bit harder than retrieval
more needles is more difficult and needles towards the start of your context are harder to retrieve than
towards the end those are three main observations from this and it maybe indeed due to this recency bias so
overall what this kind of tells you is be wary of just context stuffing in large long context
there are no retrieval guarantees and also there's some recent results that came out actually just today suggesting that single needle may
be misleadingly easy um you know there's no reasoning it's retrieving a single
needle um and also these guys I'm I showed this tweet here show that um the
in a lot of these needle and Haack challenges including mine the facts that we look for are very different than um
the background kind of Hy stack of Paul Graham essays and so that may be kind of an interesting artifact they note that
indeed if the needle is more subtle retrievals is worse so I think basically
when you see these really strong performing needle and hyack analyses put up by model providers you should be
skeptical um you shouldn't necessarily assume that you're going to get high quality retrieval from these long contact LMS uh for numerous reasons you
need to think about retrieval of multiple facts um you need to think about reasoning on top of retrieval you
need need to think about the subtlety of the retrieval relative to the background context because for many of these needle
and the Haack challenges it's a single needle no reasoning and the needle itself is very different from the
background so anyway those may all make the challenge a bit easier than a real world scenario of fact retrieval so I
just want to like kind of lay out that those cautionary notes but you know I
think it is fair to say this will certainly get better and I think it's also fair to say that rag will change
and this is just like a nearly not a great joke but Frank zap a musician made the point Jazz isn't dead it just smells
funny you know I think same for rag rag is not dead but it will change I think that's like kind of the key Point here
um so just as a followup on that rag today's focus on precise retrieval of relevant doc chunks so it's very focused
on typically taking documents chunking them in some particular way often using very OS syncratic chunking methods
things like chunk size are kind of picked almost arbitrarily embeding them storing them in an index taking a

question embedding it doing K&N uh similarity search to retrieve relevant chunks you're often setting a k
parameter which is the number of chunks you retrieve you often will do some kind of filtering or Pro processing on the
retrieve chunks and then ground your answer in those retrieved chunks so it's very focused on precise retrieval of
just the right chunks now in a world where you have very long context models
I think there's a fair question to ask is is this really kind of the most most reasonable approach so kind of on the
left here you can kind of see this notion closer to today of I need the exact relevant chunk you can risk over
engineering you can have you know higher complexity sensitivity to these odd parameters like chunk size k um and you
can indeed suffer lower recall because you're really only picking very precise chunks you're beholden to very
particular embedding models so you know I think going forward as long context models get better and better there are
definitely question you should certainly question the current kind of very precise chunking rag Paradigm but on the
flip side I think just throwing all your docs into context probably will also not be the preferred approach you'll suffer
higher latency higher token usage I should note that today 100,000 token GPD 4 is like $1 per generation I spent a
lot of money on Lang Chain's account uh on that multile analysis I don't want to tell Harrison how much I spent uh so
it's it's you know it's not good right um You Can't audit retrieve um and security and and authentication
are issues if for example you need different users different different access to certain kind of retriev
documents or chunks in the Contex stuffing case you you kind of can't manage security as easily so there's
probably some predo optimal regime kind of here in the middle and um you know I
I put this out on Twitter I think there's some reasonable points raised I think you know this inclusion at the document level is probably pretty sane
documents are self-contained chunks of context um so you know what about
document Centric rag so no chunking uh but just like operate on the context of full documents so you know if you think
forward to the rag Paradigm that's document Centric you still have the problem of taking an input question
routing it to the right document um this doesn't change so I think a lot of methods that we think about for kind of
query analysis um taking an input question rewriting it in a certain way
to optimize retrieval things like routing taking a question routing to the right database be it a relational
database graph database Vector store um and quer construction methods so for example text to SQL text to Cipher for
graphs um or text to even like metadata filters for for Vector stores those are

all still relevant in the world that you have long Contex llms um you're probably not going to dump your entire SQL DB and
feed that to the llm you're still going to have SQL queries you're still going to have graph queries um you may be more
permissive with what you extract but it still is very reasonable to store the majority of your structured data in
these in these forms likewise with unstructured data like documents like we said before it still probably makes
sense to ENC to you know store documents independently but just simply aim to retrieve full documents rather than
worrying about these idiosyncratic parameters like like chunk size um and
along those lines there's a lot of methods out there we've we've done a few of these that are kind of well optimized
for document retrieval so one I want a flag is what we call multi repesent presentation indexing and there's
actually a really nice paper on this called dense X retriever or proposition indexing but the main point is simply
this would you do is you take your OD document you produce a representation like a summary of that document you
index that summary right and then um at retrieval time you ask your question you
embed your question and you simply use a highle summary to just retrieve the right document you pass the full
document to the LM for a kind of final generation so it's kind of a trick where
you don't have to worry about embedding full documents in this particular case you can use kind of very nice
descriptive summarization prompts to build descriptive summaries and the problem you're solving here is just get
me the right document it's an easier problem than get me the right chunk so this is kind of a nice approach it
there's also different variants of it which I share below one is called parent document retriever where you could use in principle if you wanted smaller
chunks but then just return full documents but anyway the point is preserving full documents for Generation
but using representations like summaries or chunks for retrieval so that's kind of like approach one that I think is
really interesting approach two is this idea of raptor is a cool paper came out
of Stamper somewhat recently and this solves the problem of what if for certain questions I need to integrate
information across many documents so what this approach does is it takes documents and it it embeds them and
clusters them and then it summarizes each cluster um and it does this recursively in up with only one very
high level summary for the entire Corpus of documents and what they do is they take this kind of this abstraction
hierarchy so to speak of different document summarizations and they just index all of it and they use this in
retrieval and so basically if you have a question that draws an information across numerous documents you probably

have a summary present and and indexed that kind of has that answer captured so
it's a nice trick to consolidate information across documents um they they paper actually reports you know
these documents in their case or the leavs are actually document chunks or slices but I actually showed I have a
video on it and a notebook that this works across full documents as well um and this and I segue into to do this you
do need to think about long context embedding models because you're embedding full documents and that's a really interesting thing to track um the
you know hazy research uh put out a really nice um uh blog post on this
using uh what the Monch mixer so it's kind of a new architecture that tends to longer context they have a 32,000 token
embedding model that's pres that's available on together AI absolutely worth experimenting with I think this is
really interesting Trend so long long Contex and beddings kind of play really well with this kind of idea you take
full documents embed them using for example long Contex embedding models and you can kind of build these document
summarization trees um really effectively so I think this another nice trick for working with full documents in
the long context kind of llm regime um one other thing I'll note I think
there's also going to Mo be move away from kind of single shot rag well today's rag we typically you know we
chunk documents uh uh embed them store them in an index you know do retrieval
and then do generation but there's no reason why you shouldn't kind of do reasoning on top of the generation or
reasoning on top of the retrieval and feedback if there are errors so there's a really nice paper called selfrag um
that kind of reports this we implemented this using Lang graph works really well and the simp the idea is simply to you
know grade the relevance of your documents relative to your question first if they're not relevant you
rewrite the question you can do you can do many things in this case we do question rewriting and try again um we
also grade for hallucinations we grade for answer relevance but anyway it kind of moves rag from like a single shot
Paradigm to a kind of a cyclic flow uh in which you actually do various gradings Downstream and this is all
relev in the long context llm regime as well in fact you know it you you
absolutely should take advantage of of for example increasingly fast and
Performing LMS to do this grading um Frameworks like langra allow you to
build these kind of these flows which build which allows you to kind of have a more performant uh kind of kind of
self-reflective rag pipeline now I did get a lot of questions about latency here and I completely agree there's a
trade-off between kind of performance accuracy and latency that's present here I think the real answer is you can opt
to use very fast uh for example models like grock where seeing um you know gp35

turbos very fast these are fairly easy grading challenges so you can use very very fast LMS to do the grading and for
example um you you can also restrict this to only do one turn of of kind of
cyclic iteration so you can kind of restrict the latency in that way as well so anyway I think it's a really cool approach still relevant in the world as
we move towards longer context so it's kind of like building reasoning on top of rag um in the uh generation and
retrieval stages and a related point one of the challenges with rag is that your
index for example you you may have a question that is that asks something that's outside the scope of your index
and this is kind of always a problem so a really cool paper called c c rag or corrective rag came out you know a
couple months ago that basically does grading just like we talked about before and then if the documents are not
relevant you kick off and do a web search and basically return the search results to the LM for final generation
so it's a nice fallback in cases where um your you the questions out of the domain of your retriever so you know
again nice trick overlaying reasoning on top of rag I think this trend you know continues um because you know it it just
it makes rag systems you know more performant uh and less brittle to
questions that are out of domain so you know you know that's another kind of nice idea this particular approach also
we showed works really well with with uh with open source models so I ran this with mraw 7B it can run locally on my
laptop using a llama so again really nice approach I encourage you to look into this um and this is all kind of
independent of the llm kind of context length these are reasoning you can add
on top of the retrieval stage that that can kind of improve overall performance and so the overall picture kind of looks
like this where you know I think that the the the the problem of routing your
question to the right database Andor to the right document kind of remains in place query analysis is still quite
relevant routing is still relevant query construction is still relevant um in the long Contex regime I think there is less
of an emphasis on document chunking working with full documents is probably kind of more parto optimal so to speak
um there's some some clever tricks for IND indexing of documents like the
multi-representation indexing we talked
about the hierarchical indexing using Raptor that we talked about as well are two interesting ideas for document
Centric indexing um and then kind of reasoning in generation post retrieval
on retrieval itself tog grade on the generations themselves checking for hallucinations those are all kind of
interesting and relevant parts of a rag system that I think we'll probably will see more and more of as we move more
away from like a more naive prompt response Paradigm more to like a flow Paradigm we're seeing that actually

already in codenation it's probably going to carry over to rag as well where we kind of build rag systems that have
kind of a cyclic flow to them operate on documents use longc Comics llms um and still use kind of routing and query
analysis so reasoning pre- retrieval reasoning post- retrieval so anyway that was kind of my talk um and yeah feel
free to leave any comments on the video and I'll try to answer any questions but um yeah that's that's probably about it
thank you