UCLA COMPUTER SCIENCE FIELD NOTES

CS 35L

Software Construction

Git graphs, shell and Python tooling, naming contracts, React, APIs, databases, collaboration, and shipping

How these notes are written. I wrote each chapter the way I wish the material had been explained the first time: the real idea, the point where I usually get stuck, the simplest mental model that still stays honest, and a worked decision instead of a polished answer appearing from nowhere. Every chapter closes with practice and a fully written solution. If a sentence sounds impressive but does not help me solve or explain something, it does not belong here.

Daniel Mastick

Curriculum map, working notes, practice, and solutions

Course map

Week	Core thread	What should be solid by the end
1	**Shells, files, processes, and names**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
2	**Git as a directed acyclic graph**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
3	**Text, regular expressions, and data pipelines**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
4	**Python modules, packaging, and tests**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
5	**HTTP, APIs, and trust boundaries**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
6	**React state and interface architecture**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
7	**Full-stack boundaries, data, and authentication**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
8	**Collaboration, CI, and shipping a project**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
9	**Integration workshop**	Connect at least three chapters in one end-to-end problem and explain where their contracts meet.
10	**Synthesis and project review**	Audit assumptions and explain a complete solution from interface to failure handling.

Scope anchor: https://catalog.registrar.ucla.edu/course/2025/COMSCI35L?siteYear=current. The sequence is aligned to the official catalog description and expanded into a practical ten-week study plan.

How I would use this packet

Treat each numbered section as one chapter. Read the formal explanation, pause at the student interpretation, and see whether the easy version still says the same thing. Rework the example without looking, then cover the solution in the chapter practice box. The cumulative set at the end is deliberately mixed: knowledge becomes durable when I have to choose the tool instead of being told which chapter I am in.

Contents

CHAPTER 1 Shells, files, processes, and names

The idea

The shell composes programs through arguments, environment variables, redirection, pipes, exit status, and process control. Paths and names are interfaces: predictable nouns, verbs, extensions, and directory boundaries reduce cognitive load across tools.

Rebuild the chapter from first principles

The claim I need to own is this: The shell composes programs through arguments, environment variables, redirection, pipes, exit status, and process control. Paths and names are interfaces: predictable nouns, verbs, extensions, and directory boundaries reduce cognitive load across tools. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of shells, files, processes, and names. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach shells, files, processes, and names on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from the course foundations to git as a directed acyclic graph. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** The shell composes programs through arguments, environment variables, redirection, pipes, exit status, and process control.
- **Mechanism.** Paths and names are interfaces: predictable nouns, verbs, extensions, and directory boundaries reduce cognitive load across tools.
- **Boundary.** The useful test is whether I can apply shells, files, processes, and names to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when shells, files, processes, and names is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I treat every filename, command, and environment variable as part of an API another person must guess correctly.

The easy version

The shell composes programs through arguments, environment variables, redirection, pipes, exit status, and process control.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Build a pipeline that finds TODO lines in

tracked source files. The conclusion is Use git ls-files to define scope, pipe through an appropriate text search, quote patterns, and inspect the pipeline exit status. Avoid parsing ls output or scanning generated directories.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Build a pipeline that finds TODO lines in tracked source files.
Solution. Use git ls-files to define scope, pipe through an appropriate text search, quote patterns, and inspect the pipeline exit status. Avoid parsing ls output or scanning generated directories.

Practice 2. Give a short oral explanation of shells, files, processes, and names that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct shells, files, processes, and names from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a shells, files, processes, and names problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 2 Git as a directed acyclic graph

The idea

Git stores content-addressed blobs, trees, commits, and refs. A commit points to a tree and parent commits, so branches are movable labels on a DAG. Merge, rebase, revert, and reset change history in fundamentally different ways.

Rebuild the chapter from first principles

The claim I need to own is this: Git stores content-addressed blobs, trees, commits, and refs. A commit points to a tree and parent commits, so branches are movable labels on a DAG. Merge, rebase, revert, and reset change history in fundamentally different ways. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of git as a directed acyclic graph. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach git as a directed acyclic graph on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from shells, files, processes, and names to text, regular expressions, and data pipelines. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Git stores content-addressed blobs, trees, commits, and refs.
- **Mechanism.** A commit points to a tree and parent commits, so branches are movable labels on a DAG.
- **Boundary.** Merge, rebase, revert, and reset change history in fundamentally different ways.
- **Decision test.** I should be able to say when git as a directed acyclic graph is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

Once I picture branches as labels rather than folders, Git stops feeling like a bag of magic commands.

The easy version

Git stores content-addressed blobs, trees, commits, and refs.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Explain what rebase does to commit identity. The conclusion is Rebase copies a sequence onto a new parent. Because each copied commit has different parent and content metadata, it receives a new hash; shared published history therefore needs care.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Explain what rebase does to commit identity.
Solution. Rebase copies a sequence onto a new parent. Because each copied commit has different parent and content metadata, it receives a new hash; shared published history therefore needs care.

Practice 2. Give a short oral explanation of git as a directed acyclic graph that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.

Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct git as a directed acyclic graph from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a git as a directed acyclic graph problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 3 Text, regular expressions, and data pipelines

The idea

Regular expressions describe text languages; stream tools transform one record at a time. Reliable pipelines define encoding, delimiters, quoting, failure behavior, and whether input can contain spaces or newlines.

Rebuild the chapter from first principles

The claim I need to own is this: Regular expressions describe text languages; stream tools transform one record at a time. Reliable pipelines define encoding, delimiters, quoting, failure behavior, and whether input can contain spaces or newlines. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of text, regular expressions, and data pipelines. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that

I can audit it later.

4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach text, regular expressions, and data pipelines on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from git as a directed acyclic graph to python modules, packaging, and tests. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Regular expressions describe text languages; stream tools transform one record at a time.
- **Mechanism.** Reliable pipelines define encoding, delimiters, quoting, failure behavior, and whether input can contain spaces or newlines.
- **Boundary.** The useful test is whether I can apply text, regular expressions, and data pipelines to a new case and explain the assumption that makes the answer valid.

- **Decision test.** I should be able to say when text, regular expressions, and data pipelines is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I write the data contract for a pipeline before chaining commands, especially the delimiter and empty-input behavior.

The easy version

Regular expressions describe text languages; stream tools transform one record at a time.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why is a greedy .* often dangerous? The conclusion is It can consume farther than intended and backtrack heavily. Use a constrained character class, non-greedy form where supported, or parse structured data with a real parser.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why is a greedy .* often dangerous?
Solution. It can consume farther than intended and backtrack heavily. Use a constrained character class, non-greedy form where supported, or parse structured data with a real parser.

Practice 2. Give a short oral explanation of text, regular expressions, and data pipelines that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct text, regular expressions, and data pipelines from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a text, regular expressions, and data pipelines problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output?

Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 4 Python modules, packaging, and tests

The idea

Python projects need explicit module boundaries, environments, dependencies, entry points, and deterministic tests. Type hints document intent but do not replace runtime validation. Fixtures isolate repeatable setup; mocks belong at expensive or nondeterministic boundaries.

Rebuild the chapter from first principles

The claim I need to own is this: Python projects need explicit module boundaries, environments, dependencies, entry points, and deterministic tests. Type hints document intent but do not replace runtime validation. Fixtures isolate repeatable setup; mocks belong at expensive or nondeterministic boundaries. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of python modules, packaging, and tests. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.

- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach python modules, packaging, and tests on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from text, regular expressions, and data pipelines to http, apis, and trust boundaries. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Python projects need explicit module boundaries, environments, dependencies, entry points, and deterministic tests.
- **Mechanism.** Type hints document intent but do not replace runtime validation.
- **Boundary.** Fixtures isolate repeatable setup; mocks belong at expensive or nondeterministic boundaries.
- **Decision test.** I should be able to say when python modules, packaging, and tests is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I keep core logic pure and push files, clocks, networks, and environment access to thin edges so tests stay fast.

The easy version

Python projects need explicit module boundaries, environments, dependencies, entry points, and deterministic tests.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Design a testable API client. The conclusion is Inject a transport interface and clock, keep response parsing pure, and test success, timeout, malformed data, and retry limits with a fake transport rather than the live service.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Design a testable API client.
Solution. Inject a transport interface and clock, keep response parsing pure, and test success, timeout, malformed data, and retry limits with a fake transport rather than the live service.

Practice 2. Give a short oral explanation of python modules, packaging, and tests that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct python modules, packaging, and tests from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a python modules, packaging, and tests problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 5 HTTP, APIs, and trust boundaries

The idea

HTTP combines method semantics, resource identifiers, headers, status codes, caching, authentication, and representation formats. Server validation is mandatory because client code is not a security boundary. Idempotency and explicit error schemas make retries and debugging safer.

Rebuild the chapter from first principles

The claim I need to own is this: HTTP combines method semantics, resource identifiers, headers, status codes, caching, authentication, and representation formats. Server validation is mandatory because client code is not a security boundary. Idempotency and explicit error schemas make retries and debugging safer. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of http, apis, and trust boundaries. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach http, apis, and trust boundaries on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from python modules, packaging, and tests to react state and interface architecture. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** HTTP combines method semantics, resource identifiers, headers, status codes, caching, authentication, and representation formats.
- **Mechanism.** Server validation is mandatory because client code is not a security boundary.
- **Boundary.** Idempotency and explicit error schemas make retries and debugging safer.
- **Decision test.** I should be able to say when http, apis, and trust boundaries is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I design an API around resources and failure cases before writing route handlers.

The easy version

HTTP combines method semantics, resource identifiers, headers, status codes, caching, authentication, and representation formats.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Choose a status for creating a resource and for invalid input. The conclusion is Successful creation normally returns 201 with a Location or representation. Invalid client data returns 400 or 422 by the chosen contract; authentication and authorization failures remain distinct.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Choose a status for creating a resource and for invalid input.
Solution. Successful creation normally returns 201 with a Location or representation. Invalid client data returns 400 or 422 by the chosen contract; authentication and authorization failures remain distinct.

Practice 2. Give a short oral explanation of http, apis, and trust boundaries that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct http, apis, and trust boundaries from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a http, apis, and trust boundaries problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 6 React state and interface architecture

The idea

React renders UI as a function of props and state. State should have one owner, derived values should be computed rather than duplicated, and effects synchronize with external systems rather than repair ordinary data flow. Stable keys preserve component identity.

Rebuild the chapter from first principles

The claim I need to own is this: React renders UI as a function of props and state. State should have one owner, derived values should be computed rather than duplicated, and effects synchronize with external systems rather than repair ordinary data flow. Stable keys preserve component identity. I do not count

that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of react state and interface architecture. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach react state and interface architecture on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from http, apis, and trust boundaries to full-stack boundaries, data, and authentication. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** React renders UI as a function of props and state.
- **Mechanism.** State should have one owner, derived values should be computed rather than duplicated, and effects synchronize with external systems rather than repair ordinary data flow.
- **Boundary.** Stable keys preserve component identity.
- **Decision test.** I should be able to say when react state and interface architecture is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

If two pieces of state can disagree, I usually have one piece too many.

The easy version

React renders UI as a function of props and state.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why is array index a risky list key? The conclusion is After insertion or reordering, the same index may refer to different data, so React can preserve the wrong component state. Use a stable domain identifier.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why is array index a risky list key?
Solution. After insertion or reordering, the same index may refer to different data, so React can preserve the wrong component state. Use a stable domain identifier.

Practice 2. Give a short oral explanation of react state and interface architecture that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a

four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct react state and interface architecture from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a react state and interface architecture problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 7 Full-stack boundaries, data, and authentication

The idea

A full-stack system crosses browser, API, database, job, and third-party boundaries. Schemas, migrations, transactions, session rules, secrets, and authorization connect those pieces. A dependency graph should keep domain logic inward and infrastructure replaceable.

Rebuild the chapter from first principles

The claim I need to own is this: A full-stack system crosses browser, API, database, job, and third-party boundaries. Schemas, migrations, transactions, session rules, secrets, and authorization connect those pieces. A dependency graph should keep domain logic inward and infrastructure replaceable. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of full-stack boundaries, data, and authentication. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.

4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach full-stack boundaries, data, and authentication on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from react state and interface architecture to collaboration, ci, and shipping a project. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** A full-stack system crosses browser, API, database, job, and third-party boundaries.
- **Mechanism.** Schemas, migrations, transactions, session rules, secrets, and authorization connect those pieces.
- **Boundary.** A dependency graph should keep domain logic inward and infrastructure replaceable.
- **Decision test.** I should be able to say when full-stack boundaries, data, and authentication is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I draw the request path and mark every trust boundary; that shows where validation, authorization, and observability belong.

The easy version

A full-stack system crosses browser, API, database, job, and third-party boundaries.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Where should a service-role database key live? The conclusion is Only on the trusted server or controlled scripts, never in browser bundles. The client receives a narrowly scoped public credential and the server enforces authorization.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Where should a service-role database key live?
Solution. Only on the trusted server or controlled scripts, never in browser bundles. The client receives a narrowly scoped public credential and the server enforces authorization.

Practice 2. Give a short oral explanation of full-stack boundaries, data, and authentication that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct full-stack boundaries, data, and authentication from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a full-stack boundaries, data, and authentication problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the

numerical or verbal result as unverified.

CHAPTER 8 Collaboration, CI, and shipping a project

The idea

Small commits, focused branches, reviewable diffs, issue scopes, and written decisions reduce merge risk. CI should reproduce formatting, type checking, tests, builds, and security checks. Shipping also needs configuration validation, logs, rollback, and a usable README.

Rebuild the chapter from first principles

The claim I need to own is this: Small commits, focused branches, reviewable diffs, issue scopes, and written decisions reduce merge risk. CI should reproduce formatting, type checking, tests, builds, and security checks. Shipping also needs configuration validation, logs, rollback, and a usable README. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of collaboration, ci, and shipping a project. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.

- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach collaboration, ci, and shipping a project on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from full-stack boundaries, data, and authentication to the cumulative course model. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Small commits, focused branches, reviewable diffs, issue scopes, and written decisions reduce merge risk.
- **Mechanism.** CI should reproduce formatting, type checking, tests, builds, and security checks.
- **Boundary.** Shipping also needs configuration validation, logs, rollback, and a usable README.
- **Decision test.** I should be able to say when collaboration, ci, and shipping a project is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

Done means another person can clone, configure, test, understand, and recover the project without asking what I meant.

The easy version

Small commits, focused branches, reviewable diffs, issue scopes, and written decisions reduce merge risk.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The

worked question below makes that reasoning concrete: Design a minimal pull-request gate for a React/Python service. The conclusion is Run frontend lint, typecheck, tests, and production build; run Python formatting/lint, type checks, and tests; scan dependencies and secrets; require review and protect the main branch.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Design a minimal pull-request gate for a React/Python service.
Solution. Run frontend lint, typecheck, tests, and production build; run Python formatting/lint, type checks, and tests; scan dependencies and secrets; require review and protect the main branch.

Practice 2. Give a short oral explanation of collaboration, ci, and shipping a project that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct collaboration, ci, and shipping a project from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a collaboration, ci, and shipping a project problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

Final study check

I should be able to close this packet and reconstruct the core models, not merely recognize their names. My final check is to explain one complete problem aloud: what the inputs mean, which invariant or contract controls the work, why the method is legal, what it costs, how it can fail, and what evidence would convince another engineer.