ThoughtSTEM Language Assessment Handbook

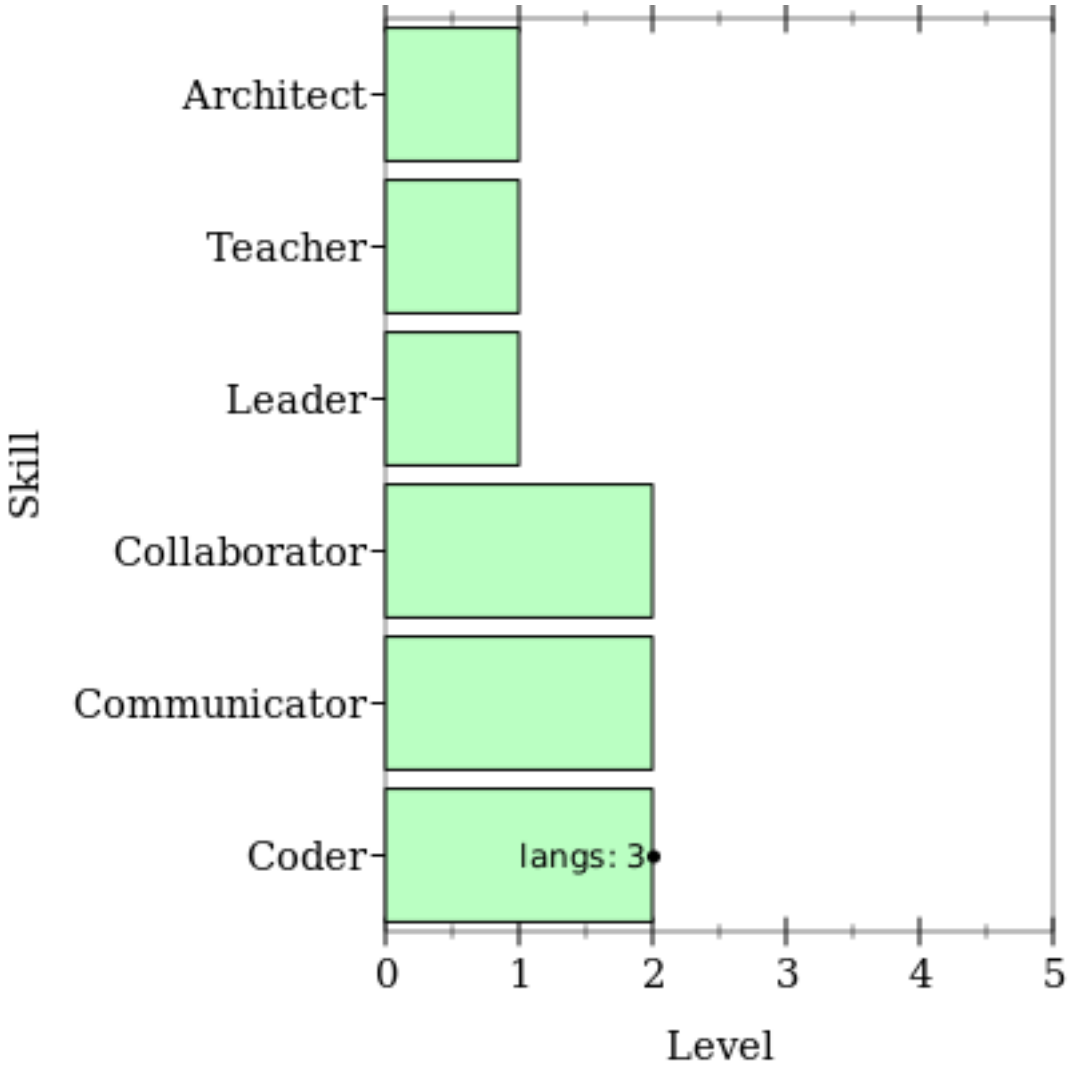
Version 7.3

June 17, 2019

1 Introduction

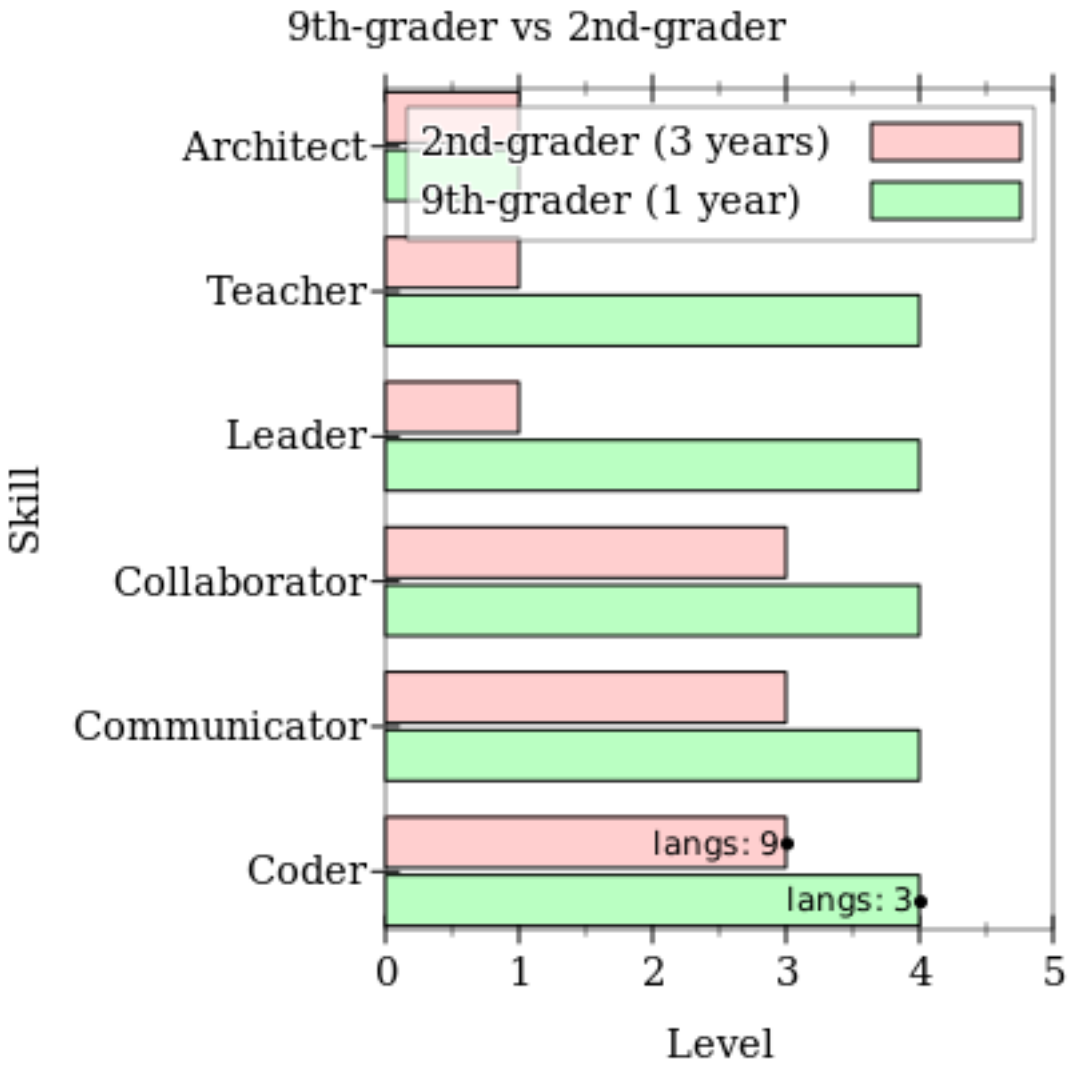
Welcome! This handbook discusses the theory and practice of assessments at Thought-STEM. It also articulates general expectations for how students should grow as software developers.

As a gentle introduction, what an assessment seeks to do is quantify the skill level of a single programmer across six different dimensions. The example assessment chart below shows a hypothetical student that is a **1 ("Novice") Architect**, **Teacher**, and **Leader**; they are a **2 ("Advanced Beginner") Coder**, **Communicator**, and **Collaborator**; and they have been assessed on 3 languages.



This document is intended to explain what such assessments charts mean, how assessments can be performed, and the science that provides the groundwork for them.

Assessments are critical for identifying a student's strengths and weaknesses. They can also be used to set expectations and to determine if students are meeting those expectations. Our assessment model has been designed to allow us to make comparisons of individuals with diverse backgrounds. For example, here is a comparative assessment of the expected skill level of an 2nd grader who has been training for 3 years to the expected skill level of a 9th grader who has been training for 1 year.



The major sections of this book cover the core concepts of assessments, step-by-step processes for performing those assessments, and some benchmark assessment charts (i.e. the skill expectations for ThoughtSTEM students after various years of training).

2 Assessments in Theory

Educational assessments always have two components:

- Theory
- Practice

First, we'll discuss the underlying theory of assessments, then we'll discuss how assessments are actually done in practice. The theory is key to actually understanding the charts resulting from the assessment – which is why we start with the theory.

2.1 The Six Roles

There are six roles that all software developers must play at various times throughout their career. Gone are the days where coders only needed to know how to code. The 21st century coder must be more than a coder:

- **Coders** are skilled at writing code – specifically with regard to translating from high-level specifications into code (called the "implementation"). Usually, that means translating from English to code. Modern coders have mastery over many languages, and many families of languages: from general purpose languages to domain specific languages. They are also skilled at learning new languages and they embrace the challenge of adopting and mastering new technologies, thriving in a world whose software ecosystems are in constant flux – where the lightning-fast release of disruptive technologies is the norm, not the exception.
- **Communicators** are **Coders** who are skilled at communicating with and about code, both with coders and non-coders. This includes one's ability to communicate visually, verbally, in writing – i.e. drawing high-level "whiteboard" diagrams of code, producing visual communication aids, speaking articulately about code and its specifications, using appropriate technical vocabulary (when necessary), and avoiding technical vocabluary (when necessary), and verbally explaining one's thought process while writing and maintaining software. Increasingly, in the 21st century, there is a new word for coders who cannot communicate. The word is: "unemployed".
- **Collaborators** are **Coders** who are also **Communicators** who have mastered additional skills, allowing them to work in teams with other **Coder/Communicators**. They know how to distribute work equally and fairly amongst team members, take direction, self-organize, and finish projects in a timely manner without over- or underworking any member of the team. They understand how to collaboratively build systems of tremendous complexity, while maintaining a unity of vision amongst fellow developers. They understand how to work together with coders who know more than

them and also with coders who know less than them. They understand how to be effective in teams whose members have diverse skillsets. They understand themselves, others, and software design well enough to enhance any team they are a part of.

- **Leaders** are **Coders** who are also masterful **Communicators** and **Collaborators**, who have additionally learned how to master the art of leading teams of other **Coders**. They know how to distribute work appropriately to others, understand the strengths and weaknesses of other team members, know how to communicate effectively and authoritatively, can resolve conflicts, and effortlessly maintain a positive, high-energy working environment. They can communicate both about code that already exists, as well as about the long-term vision for code that has not yet been written.
- **Teachers** are **Coders** who have are also excellent **Communicators**, **Collaborators**, and **Leaders**, but have also mastered an ability to make others into **Coders**, **Communicators**, **Collaborators**, and **Leaders**. This requires an ability to explain how code works, tactfully identify the strengths and weaknesses in other people's skillsets, nurture people's strengths, improve people's weaknesses, inspire others to want to improve, explain verbally and visually how to use code they've written, explain verbally and visually how to use code that they haven't written. To teach, one must be able to lead – but the difference between **Leader** and **Teacher** is: A leader can lead a team that is already full of skilled **Collaborators**; a teacher knows how to make team members into skilled **Collaborators**, and even to train new **Leaders**.
- **Architects** are expert **Coders**, **Communicators**, **Collaborators**, **Leaders**, and **Teachers**. However, they match these skills with tremendous creativity, vision, design skills, and passion for creating large, complex systems that solve problems that have not been solved before. This requires an ability to design and build large systems over a long period of time, often while working with more than one team of coders. It includes being able to build systems that help real people, to convince those people that your system can help them, and to teach them how to use your system. Above all, it requires writing clear, concise, and complete documentation of software systems for various stakeholders. Architects must so thoroughly undrstand the implementation process that they can clearly articulate both what the system will do when complete, as well as how to build it. These are the **Coders** who have learned how to maximize their impact upon the world, by maximizing the impacts of everyone around them.

2.2 Skill Levels

There are 5 skill levels:

- **1 ("Novice")**
- **2 ("Advanced Beginner")**
- **3 ("Competent")**

- **4 ("Proficient")**
- **5 ("Expert")**

Our model of expertise is a modification of the well-known Dreyfus Model. When it comes to the Dreyfus Model, it's imperative to remember that the distance between **1 ("Novice")** and **5 ("Expert")** is gigantic.

It is, in fact, much larger than most **Novice Coders** realize. This is why ThoughtSTEM uses badges and other symbols to mark student progress during class. These can be given out more regularly and can help sustain motivation better the above model.

Students can become disheartened when they realize hard truths, such as: it may take a year or more to go from **1 ("Novice")** to **2 ("Advanced Beginner")**; and that becoming an **Expert** may take many years of hard work and deliberate practice.

The assessments in this document are good for comparing students, but not always great for motivating them. This should be made clear to any student being assessed. It is important that they know that there is no shame in being assessed at **1 ("Novice")** – and that even assessments of **2 ("Advanced Beginner")** are a big deal.

To define these terms more clearly, let us try to get a sense of what **Expert** means and work backwards from there. The following, taken together, help define **Expert**, in the context of the Dreyfus model.

Note that the Dreyfus Model was developed as a general-purpose tool for discussing expertise across a wide range of fields – not just computer science.

- An **Expert** performs tasks significantly better than a **Novice**
- An **Expert** can perform *unconsciously* at a level that a **Novice** cannot achieve even with total focus and concentration. An expert chess player, for example, can play a strong game of chess while multi-tasking – easily beating a **Novice** who is trying their best. Likewise, an Expert pianist can play a complicated piece of music from memory while carrying on a converstation; whereas a **Novice** would be unable to play the piece at all.
- An **Expert** makes an extremely high level of skill look effortless and easy. Someone who is **Competent** or **Proficient** may have an extremely high level of skill, but only an **Expert** has so internalized those skills that they scarcely need to devote their working memory to it at all.
- Most computer science college graduates are *not* **Expert** coders.
- Most straight-A computer science graduates are, at best, **Proficient** coders. Many graduates are **Advanced Beginners** and **Competent** – both of which can be sufficient for obtaining a job in the modern software workforce.
- Most computer science graduates are, at best, **Competent** communicators.
- Most computer science graduates are, at best, **Advanced Beginner** collaborators.
- Most computer science graduates are, at best, **Novice** teachers, leaders, and architects.
- Some, but not all, graduates gain **Expertise** in these areas later, after working in the workforce for many years.

Students should bear this in mind – the assessment model is designed to make comparisons between people who have only just begun to learn coding, as well as incoming college students, college graduates, working programmers, senior developers, and software architects. Obviously, the range between **1 ("Novice")** and **5 ("Expert")** *has* to be vast in order to capture the massive range of human skill that we find when we examine the programmers that live on the planet Earth today.

One thing that can help make the vastness less vast is a concept we call: **Parameterized Expertise** . That is, even a **Novice** can have a degree of **Expertise** in *specific sub-domains within the larger field of **Expertise***:

- A **Novice** pianist may be able to play certain songs at an **Expert** level
- A **Novice** chess player may be able to play chess positions as well as an **Expert** (e.g. certain standard end-game and opening positions)
- A **Novice** programmer may know certain domain specific languages as well as an **Expert** does.

This **Parameterized Expertise** is easier to assess, and such assessments can be used to estimate an individual's "theoretical **Expertise**". We can gain more and more confidence about a student's level of theoretical **Expertise** the longer and more deeply we assess their **Parameterized Expertise** in various sub-domains, but we can never know the true value for certain. Presumably, if someone can perform as well as an **Expert** in all sub-domains, then one is truly an **Expert**; but exhaustively testing an **Expert** on all possible sub-domains can take an infinite amount of time.

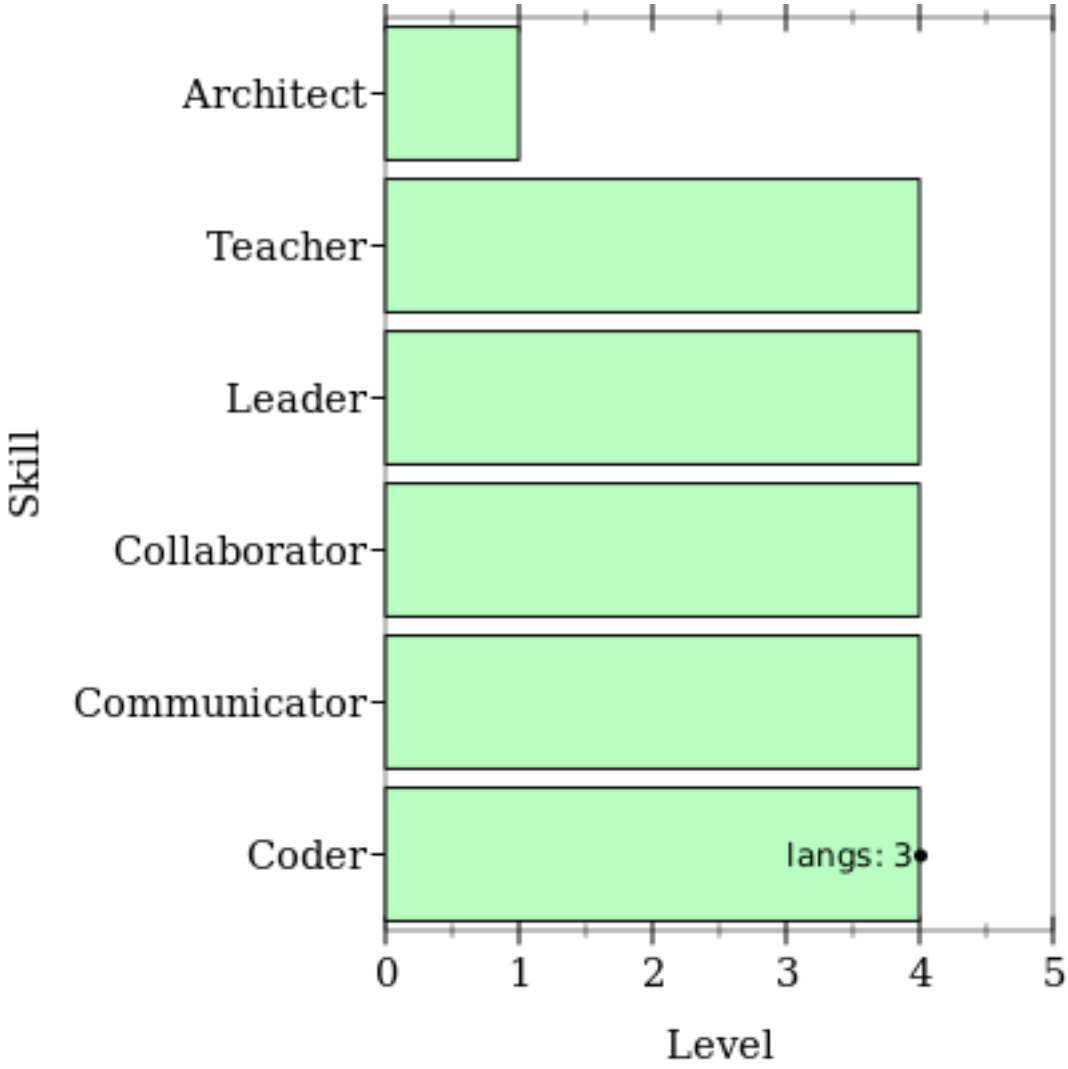
For example, if we were assessing pianists, we might choose as assessment parameters certain difficult songs, e.g.:

- **Concierto Difficulto**
- **Étude Requiring 12 Fingers**
- **Cannon in F#, A#, B#, and E# major**

Asking a pianist to learn and perform these songs is a good way to assess their skill level. But the results must always be interpreted as incomplete: Someone who performs all three songs at an **Expert** level might *only* be able to perform those three songs – which would not qualify them to be a true **Expert**, but merely an **Expert** on the chosen parameters. Still, this is better than trying to test someone on all possible songs – which would take an infinite amount of time.

To make it clear that all measures of **Expertise** are really measures of **Parameterized Expertise**, we always include when discussing assessments. They are part of the complete

story. That is why you'll always see "langs: X" printed on the bottom bar of an assessment chart:



The lang number gives a sense of how to interpret the assessment chart. An even deeper sense can be derived by examining the details of the languages on which the student was assessed.

In other words, being aware of the parameters of an assessment helps to bridge the qualitative/quantitative gap. The raw number – e.g. **3 ("Competent")** – gives us a quantitative level of expertise as an average across all of the assessed parameters. Likewise, knowledge of the parameters themselves help tell us qualitatively what the individual's expertise might

be regarding.

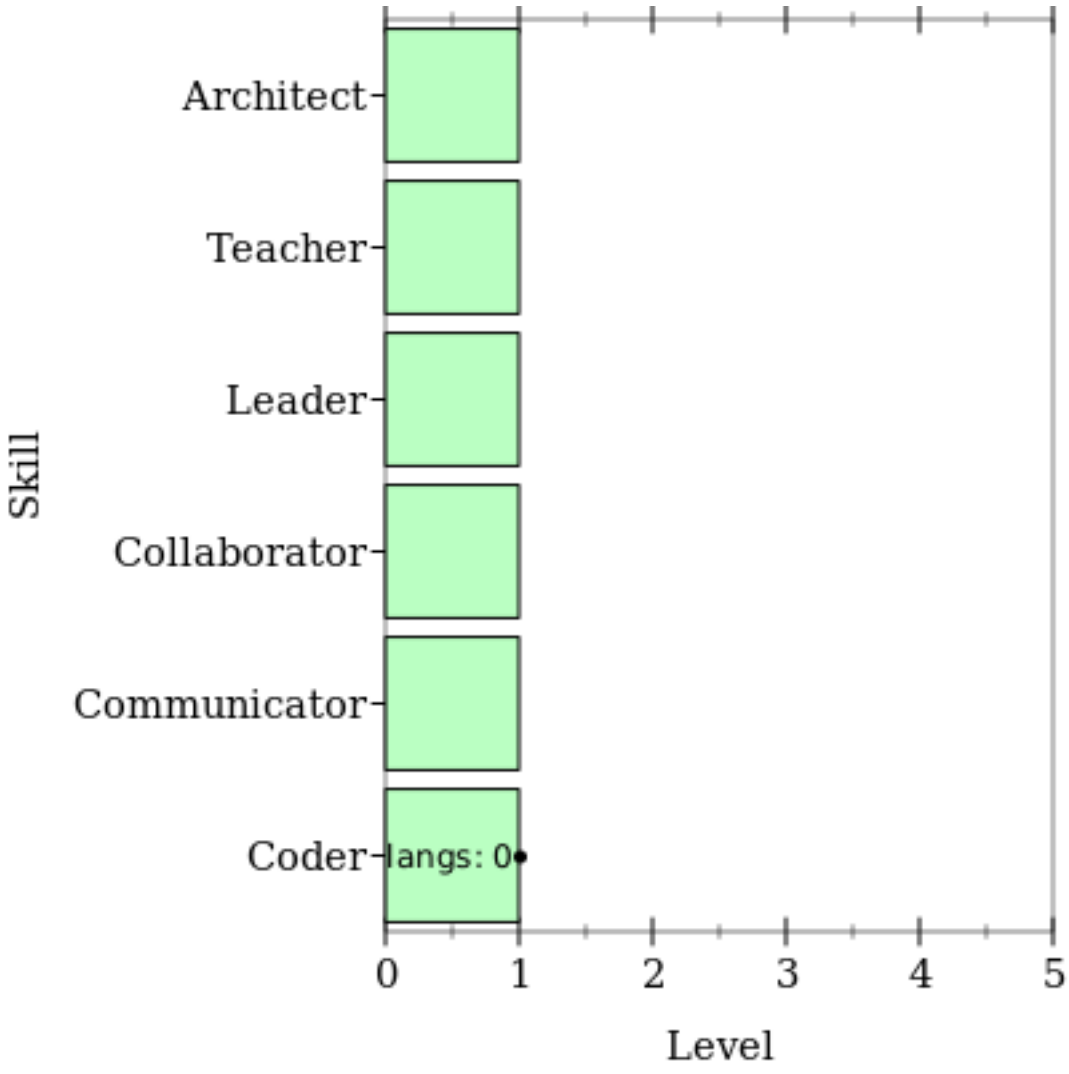
2.3 Assessment Theory, in Conclusion

To sum up the above, there is a theoretical model that provides the foundation for the **Expertise** assessments in this document.

- For any give domain (music, chess, coding, for example) there is a model of expertise known as the Dreyfus Model, which identifies 5 major milestones on the road to expert: **Novice**, **Advanced Beginner**, **Competent**, **Proficient**, **Expert** .
- The **Expert** mind can do effortlessly and intuitively what the **Novice** mind cannot, even with intense concentration.
- Assessing true **Expertise** in broad domains like "Coding" or "Piano" can be difficult; but assessing **Parameterized Expertise** can be much easier, provided that one can find suitable parameters.
- In coding, an obvious parameter is language – which is why ThoughtSTEM assessments are always in terms of **Expertise** across some number of languages.
- The assessments themselves seek to classify the assessed individual into one of the categories **Novice**, **Advanced Beginner**, **Competent**, **Proficient**, or **Expert** for some parameter language.

3 Assessments In Practice

When assessing any student for the first time, the assessment begins by assigning a **1 ("Novice")** in each of the major categories. In chart form:



Using the test algorithms in the following section, the assessor and the individual being assessed iteratively nudge the above model asymptotically toward a true assessment of the individual's expertise.

Before beginning this processes, it is important to decide how many tests to run. Considering why the assessment is being produced can assist with this. Example use-cases might be:

- **Learning conferences** – wherein students are assessed by ThoughtSTEM staff upon request (or invitation) to facilitate the co-creation of personalized learning plans.
- **Scientific publications** – to show statistically significant correlations between ThoughtSTEM pedagogical techniques and learning outcomes.

Obviously, the level of rigor in the first case can be relaxed compared to the second. To show statistical significance, more tests must be done, and the assessment process must be very formal. For learning conferences, however, it is more important to reach a consensus between parent, student, and assessor than it is to discover an absolute number. Often, consensus can be achieved simply by explaining the theory behind the assessment, coming to a mutual understanding of:

- **The 6 Skill Types**: Coder, Communicator, Collaborator, Leader, Teacher, and Architect.
- **The 5 Skill Levels**: Novice, Advanced Beginner, Competent, Proficient, and Expert.

For example, many **Advanced Beginners** with a traditional coding background (a few years of classes) will erroneously believe they are **Experts** until you describe the characteristics of an **Expert** and discuss the procedure for testing for expertise. Usually, then, students are able to modify their own definitions and appropriately categorize themselves. If a consensus cannot be reached through discussion, one or two tests is usually enough to help the student self-categorize. Once consensus is reached, the assessment can begin doing its job as a tool to aid future discussions.

The quicker an assessment can begin do its job, the better. So assessors should try to minimize the number of tests they run. With that in mind, the tests are as follows.

3.1 Assessment Test Algorithms

The high level assessment loop is as follows:

Assessor ➡ Pick a language L

UNTIL desired confidence is reached **DO**

Assessor ➡ Pick a role type to assess – Coder, Communicator, Collaborator, Leader, Teacher, Architect

Assessor ➡ go to the appropriate section of this document and run the test algorithm to get a number between 1 and 5

Assessor ➡

IF this test was the first for the current role type
this number now is the assessed skill level for the current role type
ELSE
average this number with the assessed skill level for the current role type

Tests are given in subsequent sections and are subject to change as we discover or invent new tests that take less time, require fewer resources, or are easier to administer.

3.1.1 Assessing a Coder

When assessing a **Coder**, we use a few different procedures. In most cases, the first one is sufficient for detecting **Novice**, **Advanced Beginner**, and **Competent** coders. The others can help distinguish between **Competent**, **Proficient**, and **Expert** coders.

1-card test: Novice vs Advanced Beginner Classification

UNTIL rubric produces a number **DO**

Assessor AND Coder ➡ Seat yourselves across from each other, so that the Assessor cannot see the Coder's screen but can see the Coder's face.

Assessor ➡ Randomly select 1 card from language being assessed

Assessor ➡ Read the specification aloud

Assessor ➡ give the card specification-side-up to [Coder]

Assessor ➡ say "I'd like you to implement the code from the specification written on the card. You can touch the card. You can do anything you want. There are no rules here. Just let me know when you're done, and please make sure I can see the card at all times. I will be grading you according to a rubric printed in this handbook. Would you like to review it before we begin?"

Assessor AND Coder ➡ discuss the rubric as necessary

Coder ➡ implement code from the given specification

Assessor ➡ Evaluate according to rubric

The rubric is simple:

- If the student looks at the code on the back of the card, the assessment is automatically **1 (Novice)**.
- If the student produces code that matches the specification but takes longer than 10 minutes, the assessment is **2 (Advanced Beginner)**
- If the code at the end does not match the specification, the test must be thrown out. A new card must be chosen.
- If the student produces code that matches the specification but takes less than 10 minutes, the test must be thrown out. Do the **3-card** test next time.

3-card test: Competent vs Proficient vs Expert Classification

UNTIL confidence is reached **DO**

Assessor AND Coder ➡ Seat yourselves across from each other, so that the Assessor cannot see the Coder's screen but can see the Coder's face.

Assessor ➡ Randomly select three cards from language being assessed

Assessor ➡ give the cards specification-side-up to [Coder]

Assessor ➡ say "I'd like you to write a single program that matches all three specifications. You can touch the cards. You can do anything you want. There are no rules here. Just let me know when you're done, and please make sure I can see the cards at all times. I will be grading you according to a rubric printed in this handbook. Would you like to review it before we begin?"

Assessor AND Coder ➡ discuss the rubric as necessary

Assessor ➡ say "If you are demonstrating Expertise via the Effortless Talking Rule, you may begin speaking any time before your fingers touch the keyboard."

Coder ➡ implement code from the given specifications. (If a feature of one specification conflicts with another such that you cannot do both features, you may choose which feature to implement.)

Assessor ➡ Grade according to rubric

The rubric is:

- If the student looks at the code on the back of any cards, the assessment is automatically **1 (Novice)**. The next test should be the **1-card** test.
- If the student produces code that matches the specifications but takes longer than 10 minutes, the assessment is **3 (Competent)**
- If the code at the end does not match the specifications, the test must be thrown out. The next test should be the **1-card** test.
- If the student produces code that matches the specifications in less than 10 minutes, the "Effortless Talking Rule" is applied to distinguish between **Proficient** and **Expert**.
- **The Effortless Talking Rule:** The assessment is **Expert** if the student can make it look easy and effortless to both talk and code at the same time. The talking can be about anything; however, it is most common (and perfectly acceptable) for the student to explain what they are doing while they are doing it – i.e. thinking aloud. Note that the talking must be at a comfortable conversational pace, without lulls, sentence fragments, evidence of vocal stress, or other such evidence of cognitive load. The Effortless Talking Rule tests whether or not the student has achieved automatic, effortless, and fluent recall of the target language – such that they scarsely need to think about it at all.

Note that students never "lose points" for making coding errors during the test. Experts who are demonstrating expertise are free to make mistakes and even to talk about them as they are fixing them – as long as they can do so effortlessly, and as long as they can deliver a correct program within 10 minutes. You're not grading their coding process; you are grading their cognitive load, as evidenced by their ability to verbally multitask while coding.

Note that this makes ThoughtSTEM assessments considerably more rigorous than assessments in traditional education. College-level computer science classes almost exclusively assess students on their output – not on how effortlessly the student produced that output. This leaves a major gap in the assessment model currently used in higher education. At ThoughtSTEM we close that gap when assessing **Coders** .

3.1.2 Assessing a Communicator

Note that **Communicators** must be assessed in pairs. The reason for this should be obvious. Each of the following **Communicator** tests involve communication with a partner. An individual being assessed (which we'll refer to as `Communicator-A`) may bring a partner, or the assessor may supply one (or take on the role of the partner).

In many cases, the evaluation criteria is whether both partners' code, at the end of the test, is **isomorphic**.

In cases where high accuracy is important – e.g. for scientific publications – the individual should be assessed with many **Communicators** across a range of **Expertise** levels.

Two pieces of code are **isomorphic** if:

- They both match the same specification.
- Either could be transformed into the other simply by renaming variables and functions.

1-card test: Novice vs Advanced Beginner Classification

UNTIL rubric produces a number **DO**

Communicator-B AND Communicator-A ➡ Sit facing away from each other, close enough that you can hear each other easily, but you should not be able to see each other or each other's computers.

Assessor ➡ Randomly select 1 card from language being assessed

Assessor ➡ Give the card to [Communicator-A]

Assessor ➡ say "I'd like you both to implement the code from the specification written on the card. You may both communicate at any time, using any spoke words you wish – but you may not use any means of visual communication. I will be grading your verbal communication according to the rubric in this handbook. Would you like to review it with me before we begin?"

Assessor AND Communicators ➡ discuss the rubric as necessary

Communicator-A ➡ implement code from the given specification and communicate with your partner at will.

Assessor ➡ Evaluate according to rubric

The rubric is as follows:

- If either individual ever dictates individual characters to be typed (e.g. "type a parenthesis"), the assessment for both of them is automatically **1 ("Novice")**
- If the pair produces **isomorphic** code that matches the specification, but they take longer than 10 minutes, the assessment is **2 ("Advanced Beginner")**
- If if the code at the end does not match the specification or the code on each computer is not **isomorphic**, the test must be thrown out. A new card must be chosen.
- If the pair produces code that matches the specification and is **isomorphic** but takes less than 10 minutes, the test must be thrown out. Do the **3-card** test next time.

3-card test: Competent vs Proficient vs Expert Classification

UNTIL rubric produces a number **DO**

Communicator-B AND Communicator-A ➡ Sit facing away from each other, close enough that you can hear each other easily, but you should not be able to see each other or each other's computers.

Assessor ➡ Randomly select 3 cards from language being assessed

Assessor ➡ Give the cards specification-side-up to [Communicator-A]

Assessor ➡ say "I'd like you both to implement the code from the specification written on the card. You may both communicate at any time, using any spoke words you wish – but you may not use any means of visual communication. I will be grading your verbal communication according to the rubric in this handbook. Would you like to review it with me before we begin?"

Assessor AND Coder ➡ discuss the rubric as necessary

Communicator-A ➡ implement code from the given specification and communicate with your partner at will.

Assessor ➡ Evaluate according to rubric

Code communication is a broad-spectrum skill that involves the ability to communicate about specifications, implementations, and running programs in a variety of ways: verbally, visually, and in writing. We will add more tests here if and when it becomes important to be able to assess other parts of this communication spectrum. For now, we leverage the fact that fluent verbal communication is a strong indicator of the ability to communicate in other ways.

3.1.3 Assessing a Collaborator

At ThoughtSTEM, we currently do not run scientific assessments on **Collaborators**, **Leaders**, **Teachers**, or **Architects** – due to the length of time, and number of people, such assessments would take. Capturing and observing these high-level industry skills in a laboratory setting has been a vexing issue in the history of computer science education research.

However, the theory behind ThoughtSTEM assessments can still serve as a crucial communication tool for discussing how students can calibrate their ability to self-assess their growth as as **Collaborator**, **Leader**, **Teacher**, or **Architect** .

To facilitate such discussions and/or the design of future assessments, we provide below the definitions of **Collaborator**, **Leader**, **Teacher**, and **Architect** – paired with a brief discussion of how to assess (or self-assess) these skills.

> ***Definition:*** ***Collaborators*** *are* ***Coders*** *who are also* ***Communicators*** *who have mastered additional skills, allowing them to work in teams with other* ***Coder/Communicators****. They know how to distribute work equally and fairly amongst team members, take direction, self-organize, and finish projects in a timely manner without over- or under- working any member of the team.*

They understand how to collaboratively build systems of tremendous complexity, while maintaining a unity of vision amongst fellow developers. They understand how to work together with coders who know more than them and also with coders who know less than them. They understand how to be effective in teams whose members have diverse skillsets. They understand themselves, others, and software design well enough to enhance any team they are a part of.

Why is this hard to assess? It all comes down to basic math. There are many languages – which already makes it difficult to assess someone's skill as a **Coder** . When assessing a **Communicator**, the assessment becomes trickier – due to the fact that the **Communicator** must be assessed with a partner **Communicator** (and there can be many possible partner **Communicators**). But when assessing a **Collaborator**, there are *even more* parameters – that is, the various teams of **Communicators** the individual might be placed on. Someone who excels in a team of **Expert Communicators** might flounder in a team of **Advanced Beginner Communicators** or a mixed team of **Expert Communicators** and **Novice Communicators**, and so on. Likewise, a student might excel in teams of two but struggle in teams of four.

A truly **Expert** collaborator, by definition, can enhance *any* team they are a part of, large or small, skilled or unskilled, or mixed. This means that a good laboratory assessment would require placing the individual on many different teams and running multiple assessments. This is nearly impossible to do in a lab, due to the number of people required.

However, it is much easier for a student to do in a ThoughtSTEM class – where they are routinely being placed in different teams with diverse members.

Discussion Topic: How can a ThoughtSTEM student actively seek out growth opportunities as a **Collaborator** in class?

3.1.4 Assessing a Leader

Definition: Leaders *are* ***Coders*** *who are also masterful* ***Communicators*** *and* ***Collaborators****, who have additionally learned how to master the art of leading teams of other* ***Coders****. They know how to distribute work appropriately to others, understand the strengths and weaknesses of other team members, know how to communicate effectively and authoritatively, can resolve conflicts, and effortlessly maintain a positive, high-energy working environment. They can communicate both about code that already exists, as well as about the long-term vision for code that has not yet been written.*

This is hard to assess, for the same reason that **Collaborators** are hard to assess. There are many kinds of teams, and to assess the true **Expertise** of a **Leader**, they need to be assessed on many teams.

Still, students in ThoughtSTEM classes are naturally presented with many opportunities for leadership. Ambitious students who wish to grow as **Leaders** more quickly can be encouraged to seek out more such opportunities, actively guiding the growth of their own

leadership skills.

Discussion Topic: How can a ThoughtSTEM student actively seek out Leader opportunities as a **Leader** in class?

3.1.5 Assessing a Teacher

> ***Definition: Teachers*** *are* ***Coders*** *who have are also excellent* ***Communicators***, ***Collaborators***, *and* ***Leaders***, *but have also mastered an ability to make others into* ***Coders***, ***Communicators***, ***Collaborators***, *and* ***Leaders***. *This requires an ability to explain how code works, tactfully identify the strengths and weaknesses in other people's skillsets, nurture people's strengths, improve people's weaknesses, inspire others to want to improve, explain verbally and visually how to use code they've written, explain verbally and visually how to use code that they haven't written. To teach, one must be able to lead – but the difference between* ***Leader*** *and* ***Teacher*** *is: A leader can lead a team that is already full of skilled* ***Collaborators****; a teacher knows how to make team members into skilled* ***Collaborators***, *and even to train new* ***Leaders***.

Discussion Topic: How can a ThoughtSTEM student actively seek out Teacher opportunities as a **Teacher** in class?

3.1.6 Assessing an Architect

> ***Definition: Architects*** *are expert* ***Coders***, ***Communicators***, ***Collaborators***, ***Leaders***, *and* ***Teachers***. *However, they match these skills with tremendous creativity, vision, design skills, and passion for creating large, complex systems that solve problems that have not been solved before. This requires an ability to design and build large systems over a long period of time, often while working with more than one team of coders. It includes being able to build systems that help real people, to convince those people that your system can help them, and to teach them how to use your system. Above all, it requires writing clear, concise, and complete documentation of software systems for various stakeholders. Architects must so thoroughly undrstand the implementation process that they can clearly articulate both what the system will do when complete, as well as how to build it. These are the* ***Coders*** *who have learned how to maximize their impact upon the world, by maximizing the impacts of everyone around them.*

Architect skills are the most advanced coding skills – a level to which most developers aspire, but many never reach. For students who are *extremely* ambitious, there are ways to begin acquiring such skills, at a young age. However, this can only happen if students are willing to work both in and outside of class.

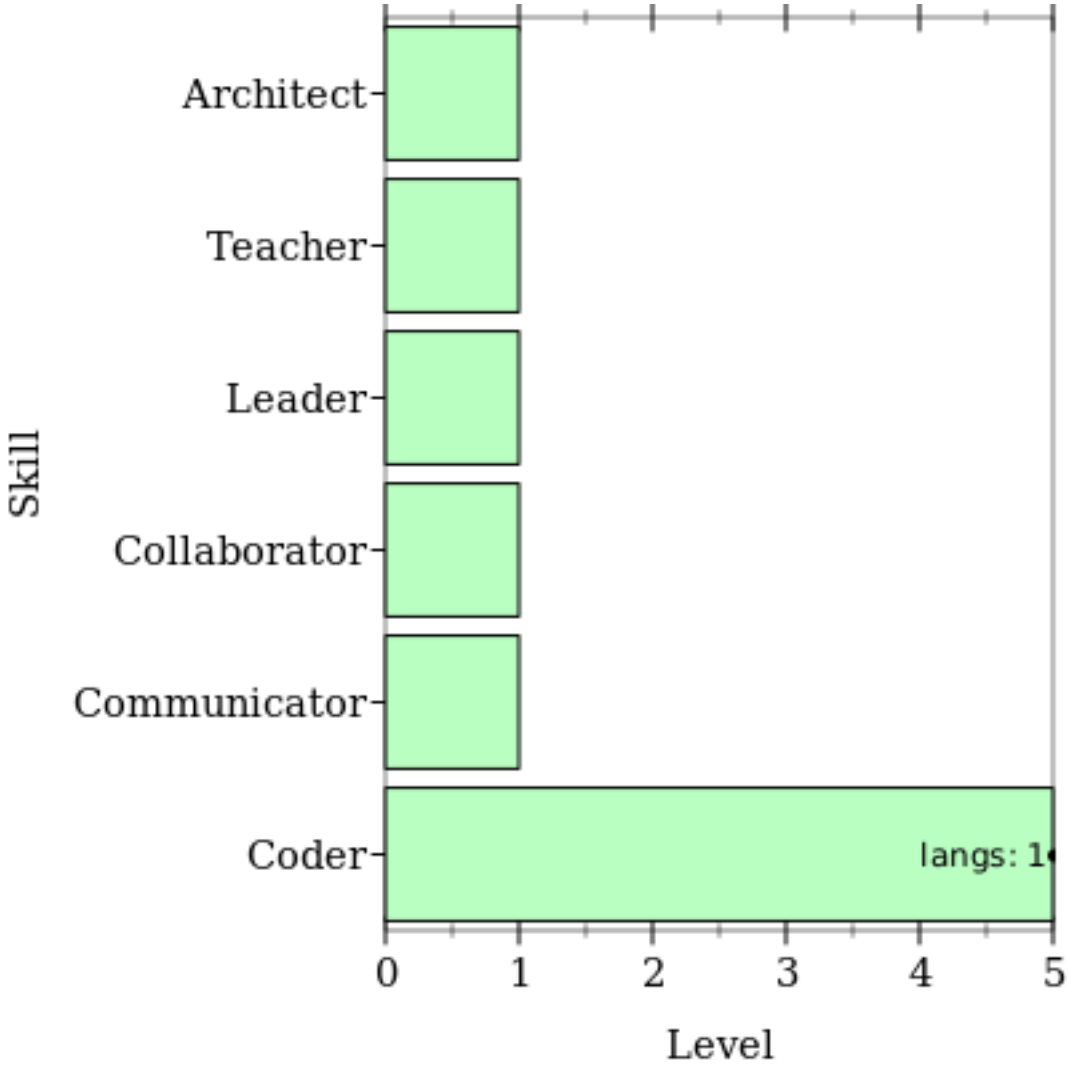
Discussion Topic: How can a ThoughtSTEM student actively seek out Architect opportunities as an **Architect** in class? outside of class?

3.2 Assessment Benchmarks

Historically, coding education has focused exclusively on coding. In a few short decades since the rise of the internet, however, the discipline of software development has become

intensely collaborative – with greater and greater communication expectations placed on working programmers. This has left education in a tricky position – where many educators (even at the college level) are literally training students for the wrong job.

The root of the problem is that many teachers and students erroneously believe that their goal is to create **Coders** that look like this:



This is the ficticious assessment chart of a supposedly "expert" coder – who knows one language well, but who has no formal training in any other skills. There are two major problems with this instructional philosophy:

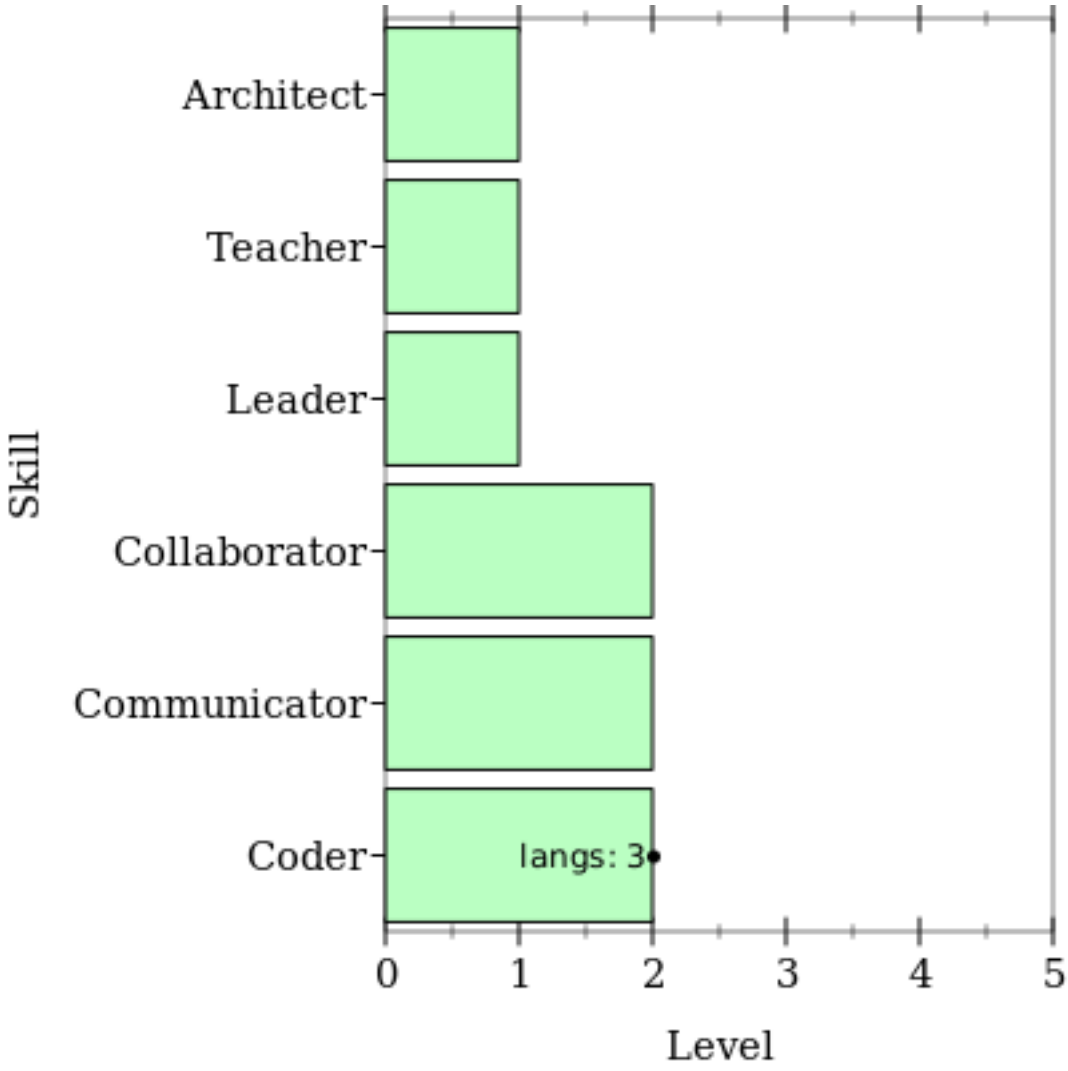
- **First, it's inefficient.** There's a wealth of scientific research that shows that people, across almost every field of study, learn faster when they learn to communicate, collaborate, lead, and teach. (The National Research Council's report "How People Learn" is a fantastic resource on the subject of how the science learning can be leveraged to design better educational environments.)
- **Second, it's immoral.** Attempting to train students in this way hurts them and everyone they will ever work with for the rest of their careers (until they finally, hopefully acquire the skills on their own). It stunts cognitive growth and is terrible for the economy. Collaborative skills are simply too important to withhold. The same is true for the ability to learn and master many languages, not just one.

Depending on the age of the student, our expectations are different – but the fundamental philosophy is the same: To train **Coders**, **Communicators**, **Collaborators**, **Leaders**, **Teachers**, and **Architects** .

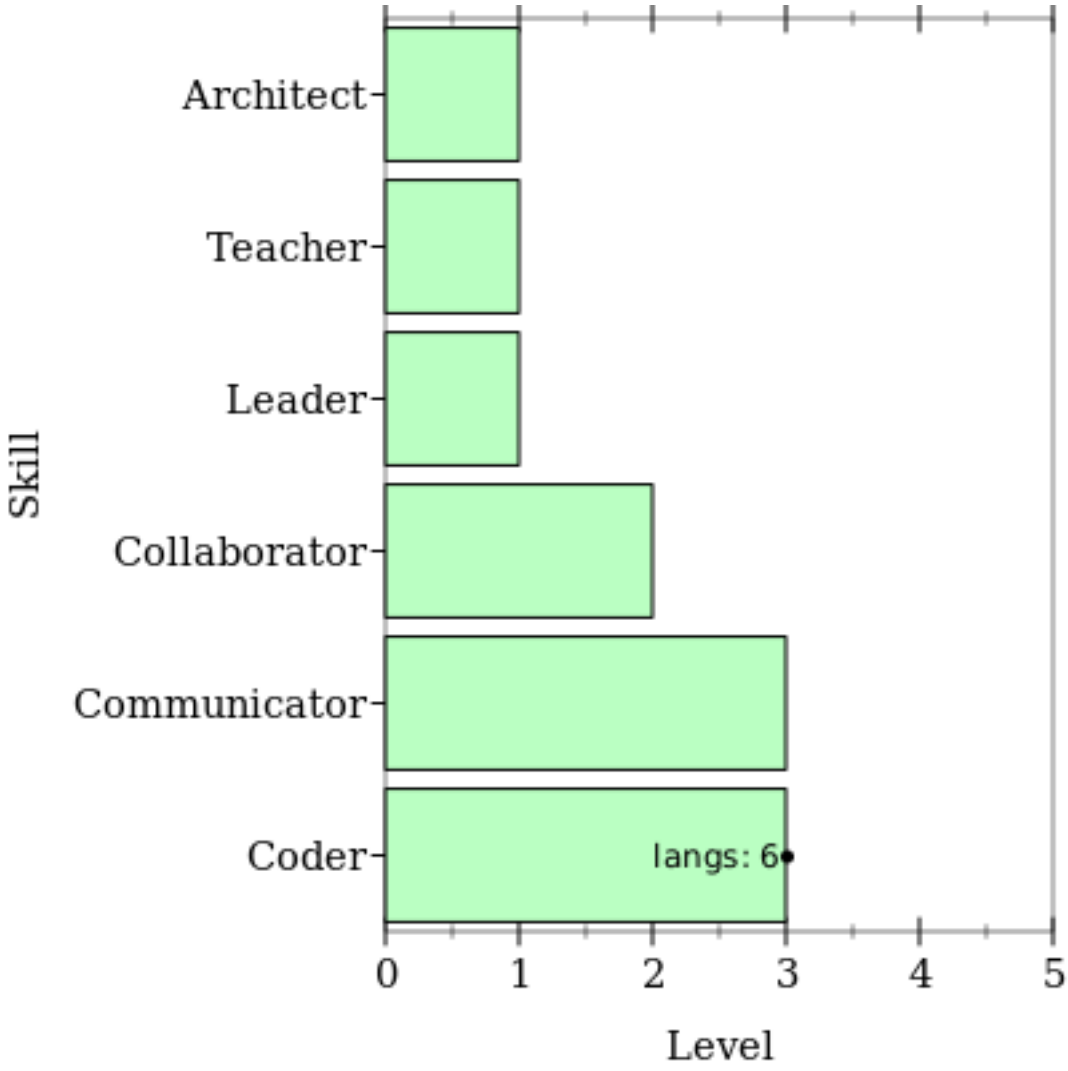
3.3 Expectations: Kindergarten through 2nd Grade

For example, a student who begins in Kindergarten and trains for one year is expected to have the following levels:

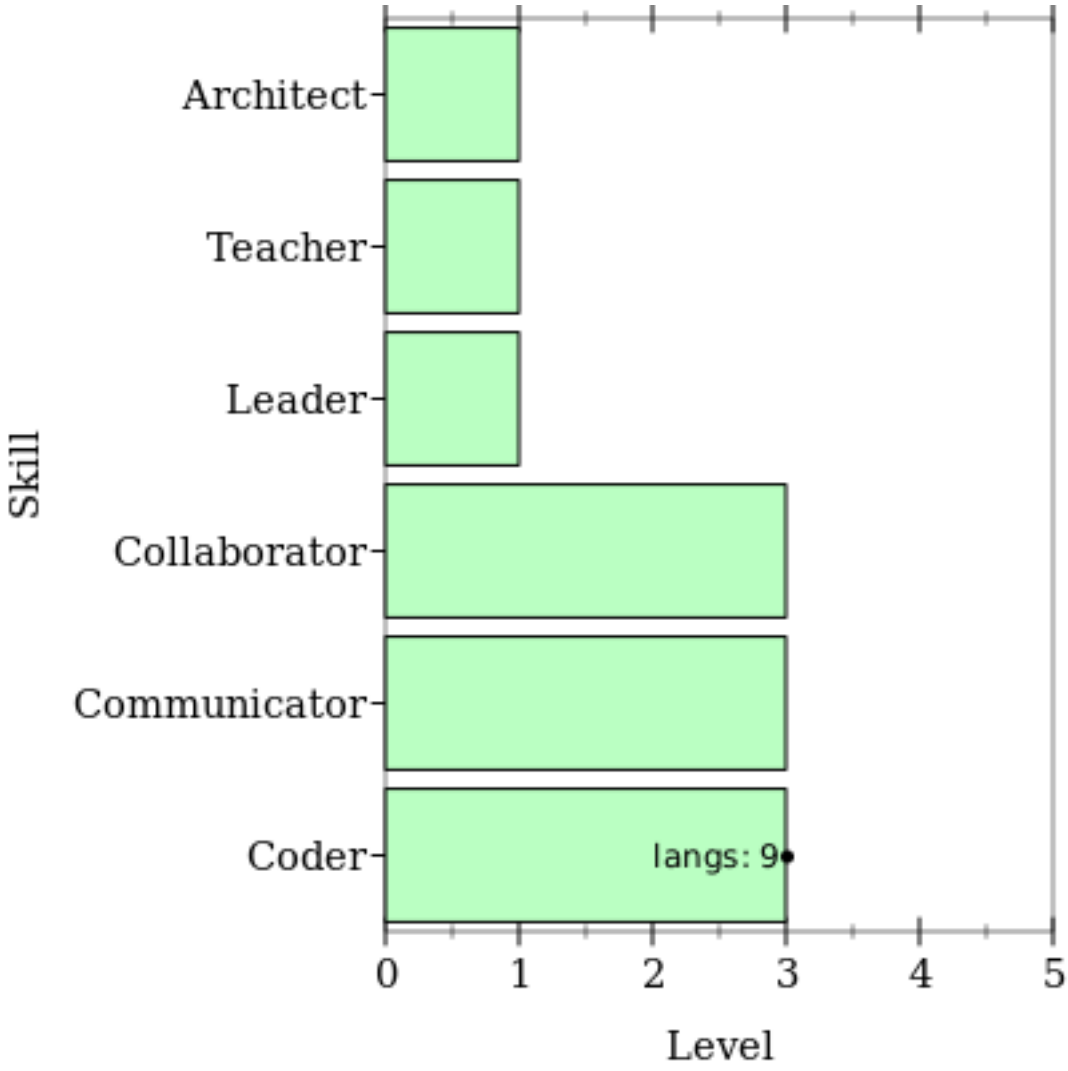
Disclaimer: These [illegible] year's classes. Many students have been taking the Thought STEM classes discussed here a year – with minimal work outside of class.



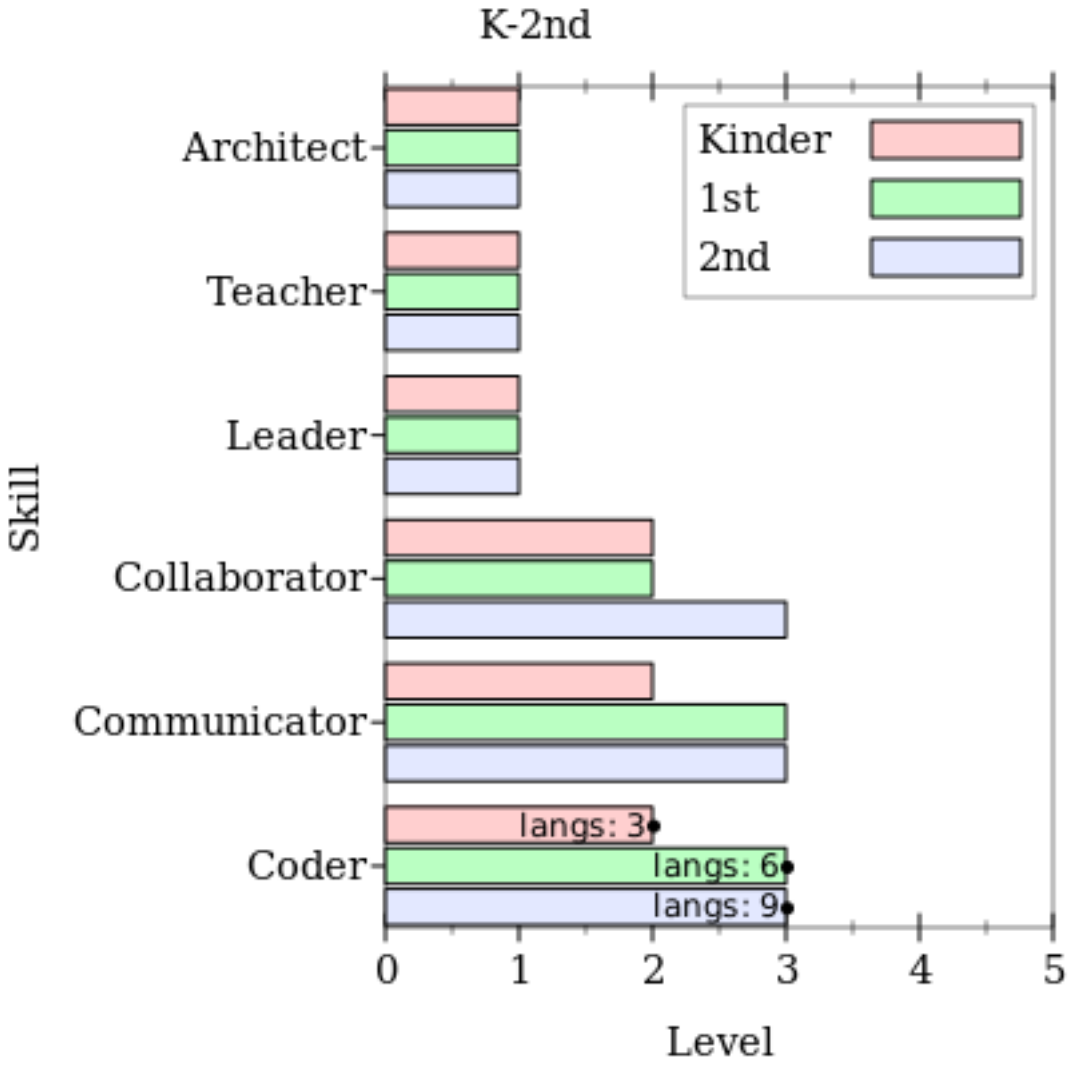
After another year, we expect students to have mastered additional languages and continued to gain **Expertise** as **Coder / Communicators** .



Finally, after another year, we expect students to have mastered additional languages and to have closed the gap between their **Collaborator** skills and their **Communicator** skills.



We can see the expected trend for the above hypothetical student better if we compare all three years on one chart:

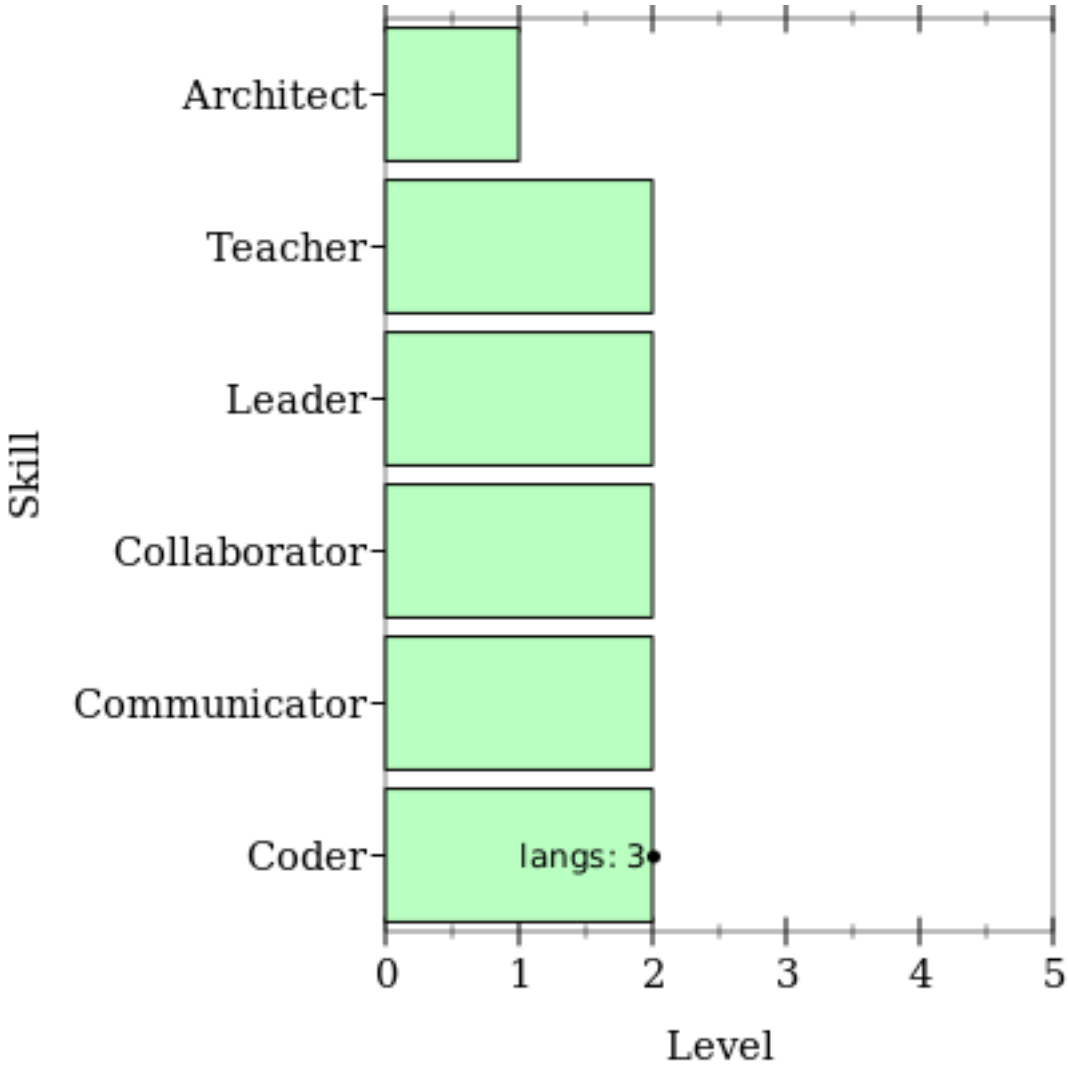


3.4 Expections: 3rd through 5th Grade

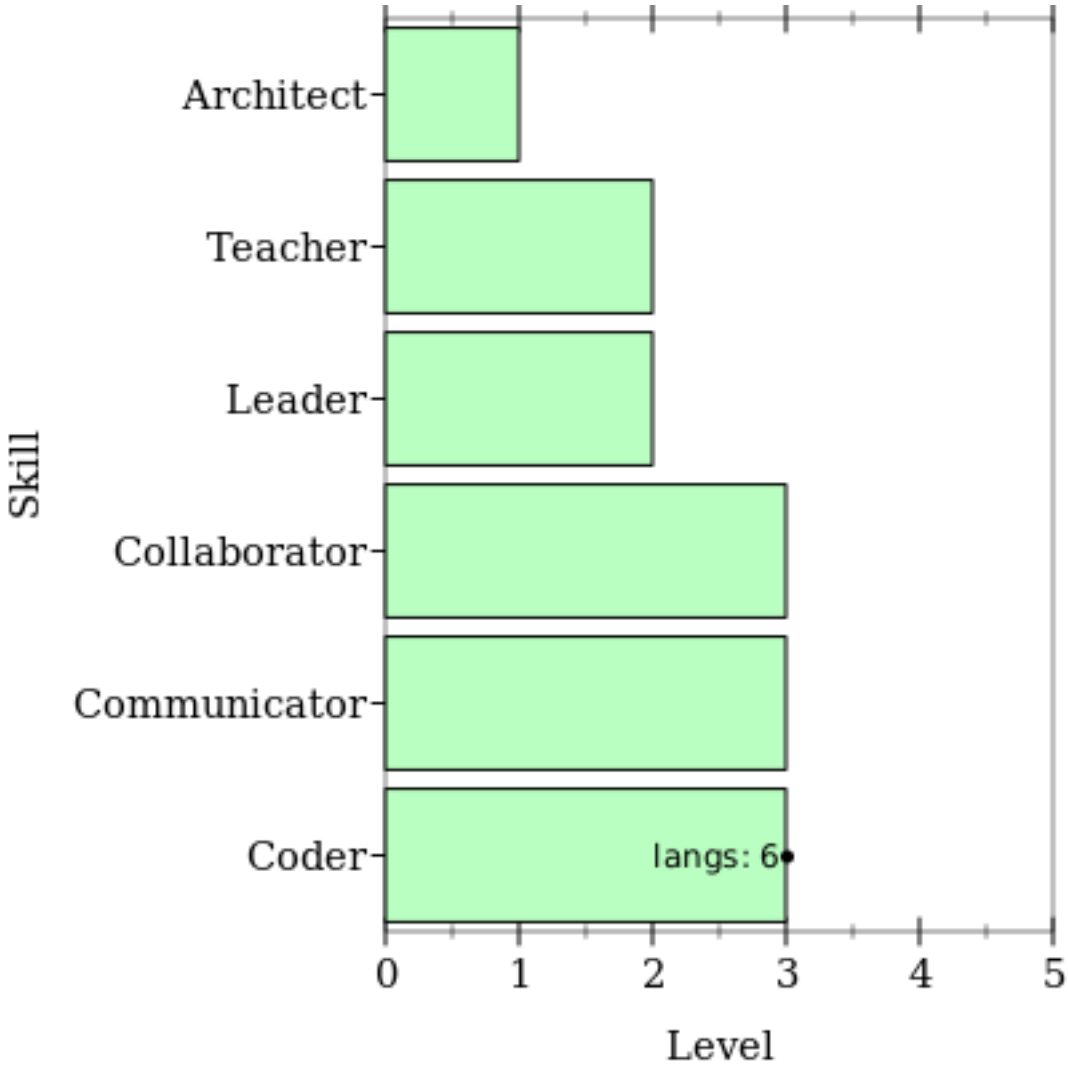
For 3rd through 5th graders (compared with younger ages), we expect that they will learn faster, and that they will have achieved sufficient cognitive development to be able to focus on higher level coding skills.

By the end of 3rd grade:

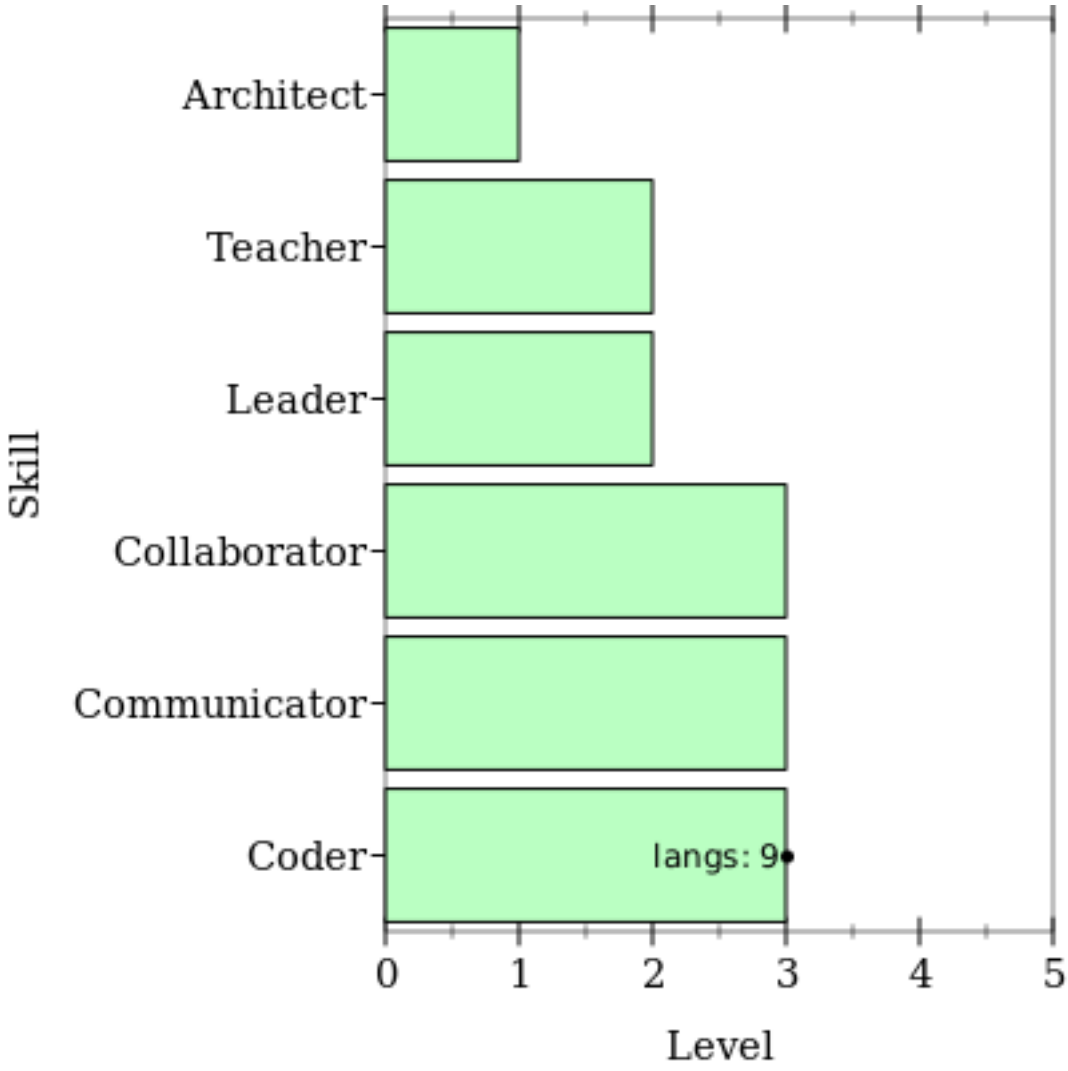
Disclaimer: These [illegible] year's lessons. Many students have already taken [illegible] Through STEM [illegible] here.



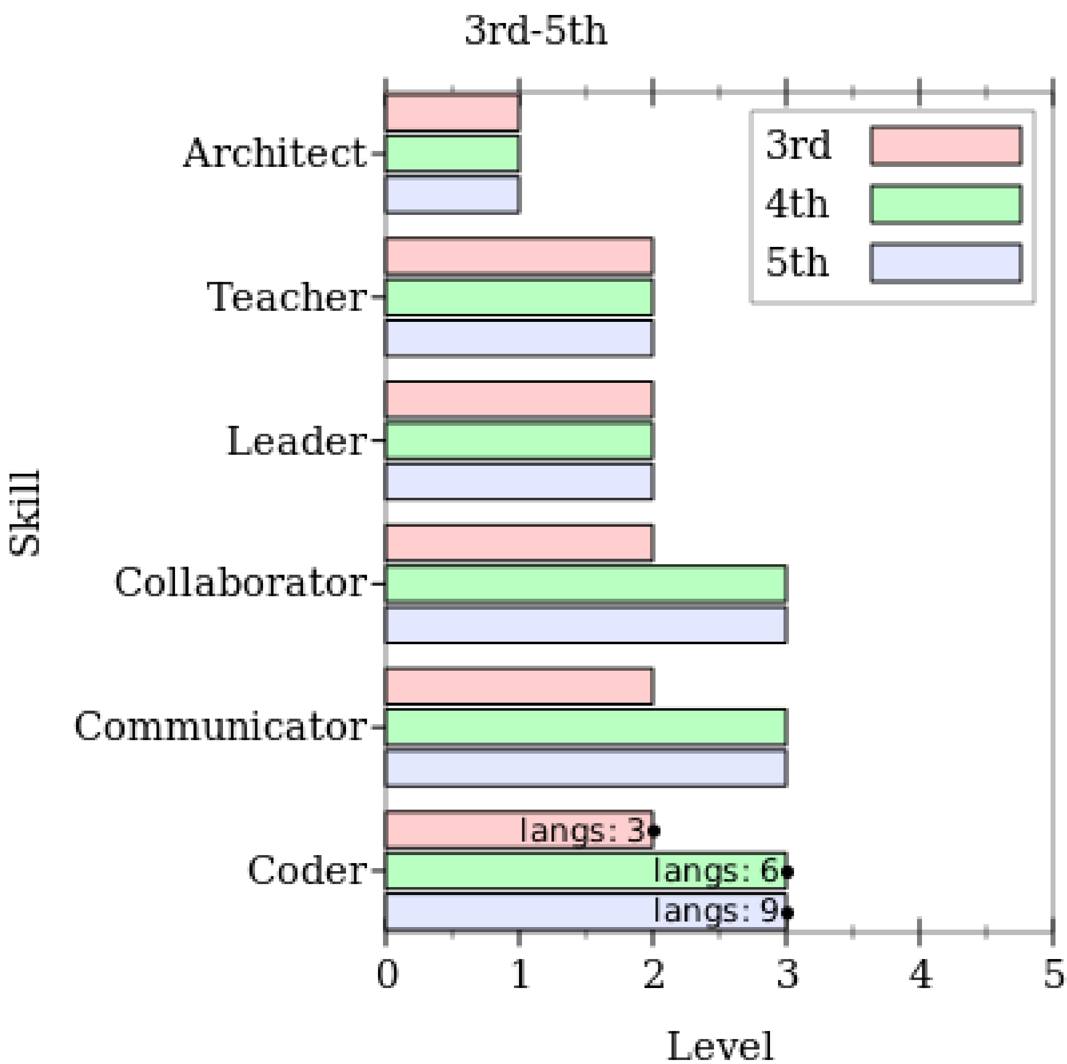
By the end of 4th grade:



By the end of 5th grade:



Comparison across all three years:

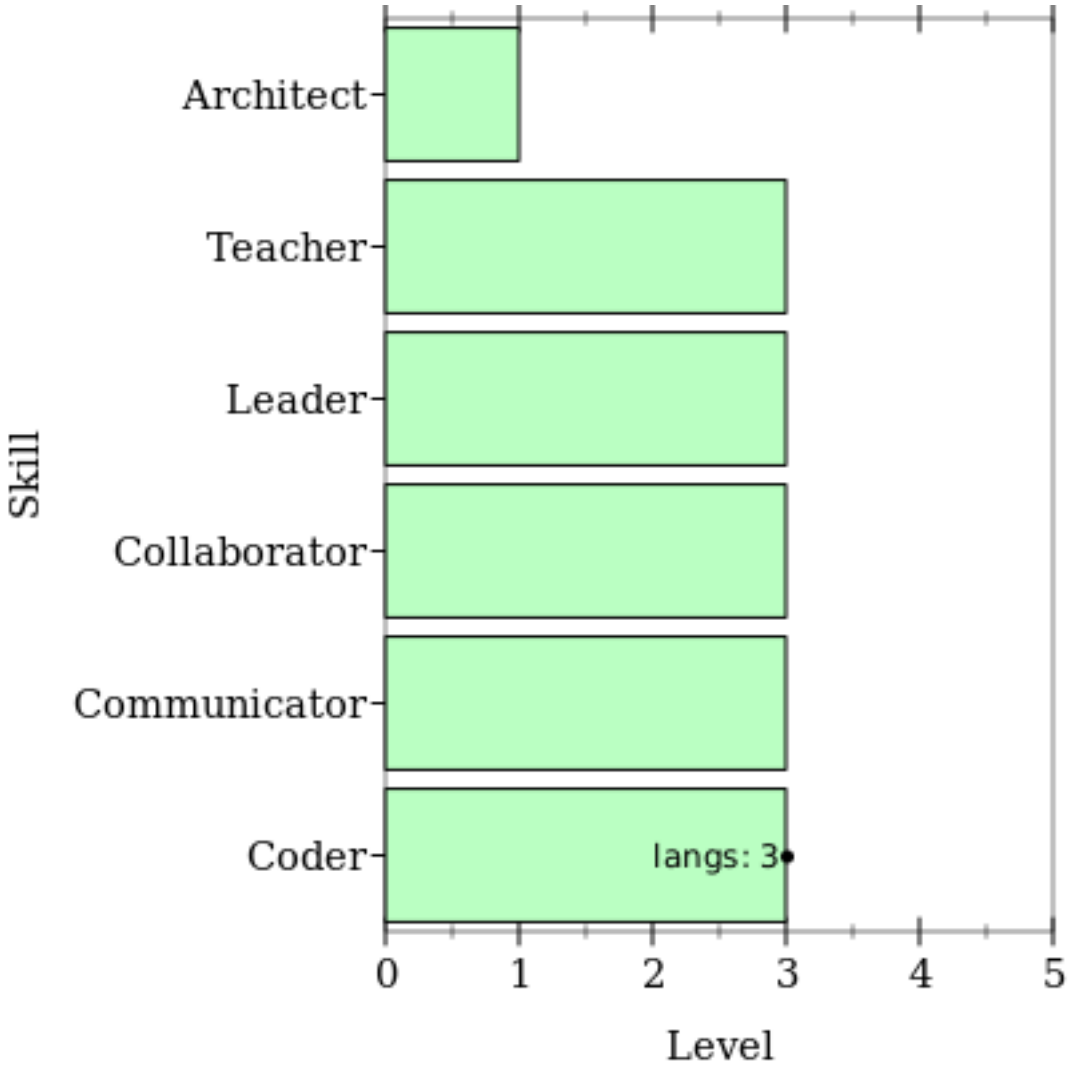


3.5 Expectations: 6th through 8th grade

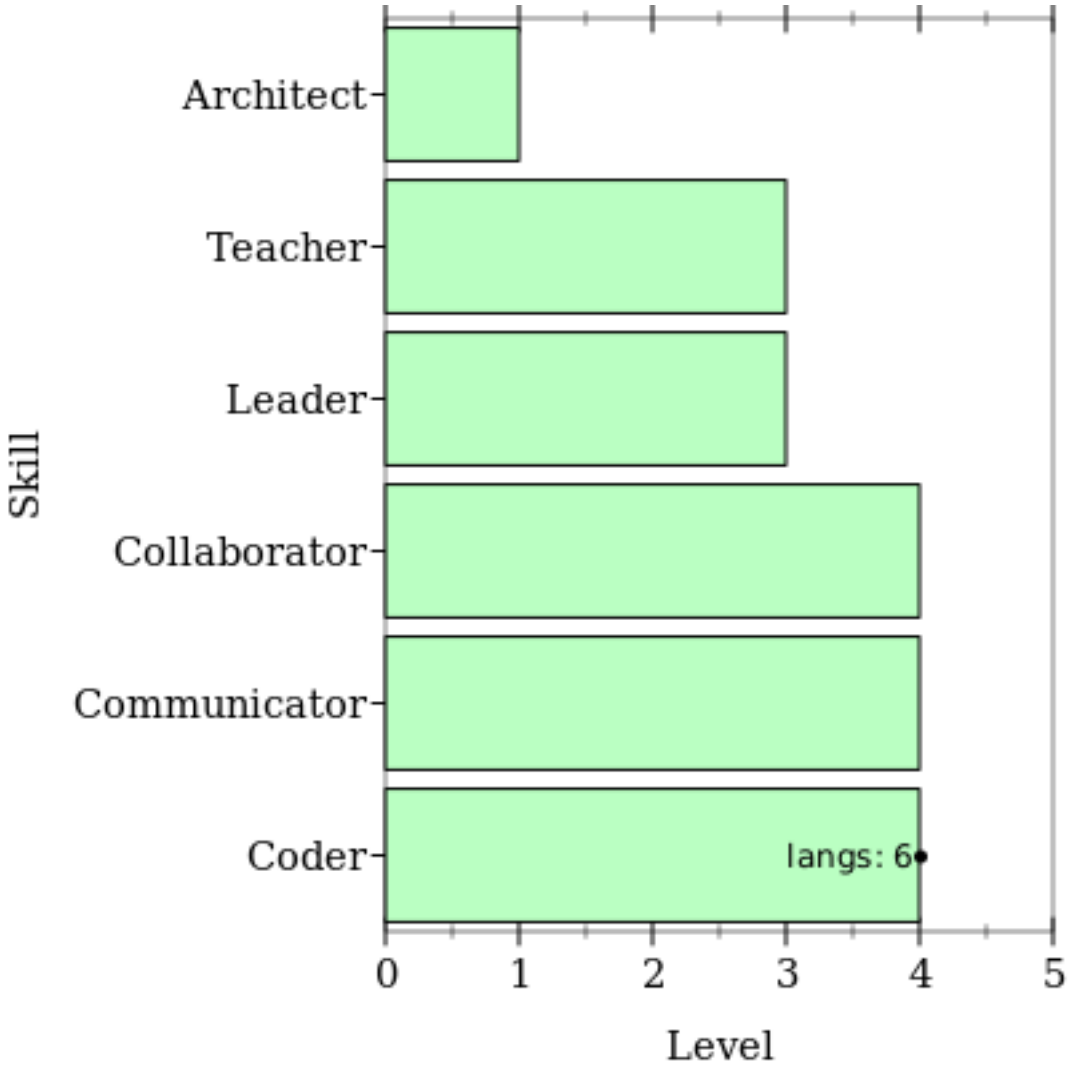
At the 6th through 8th grade level (compared with younger ages), we expect significantly more growth as **Leaders** and **Teachers**, as well as greater overall growth at the end of the three years.

By the end of 6th grade:

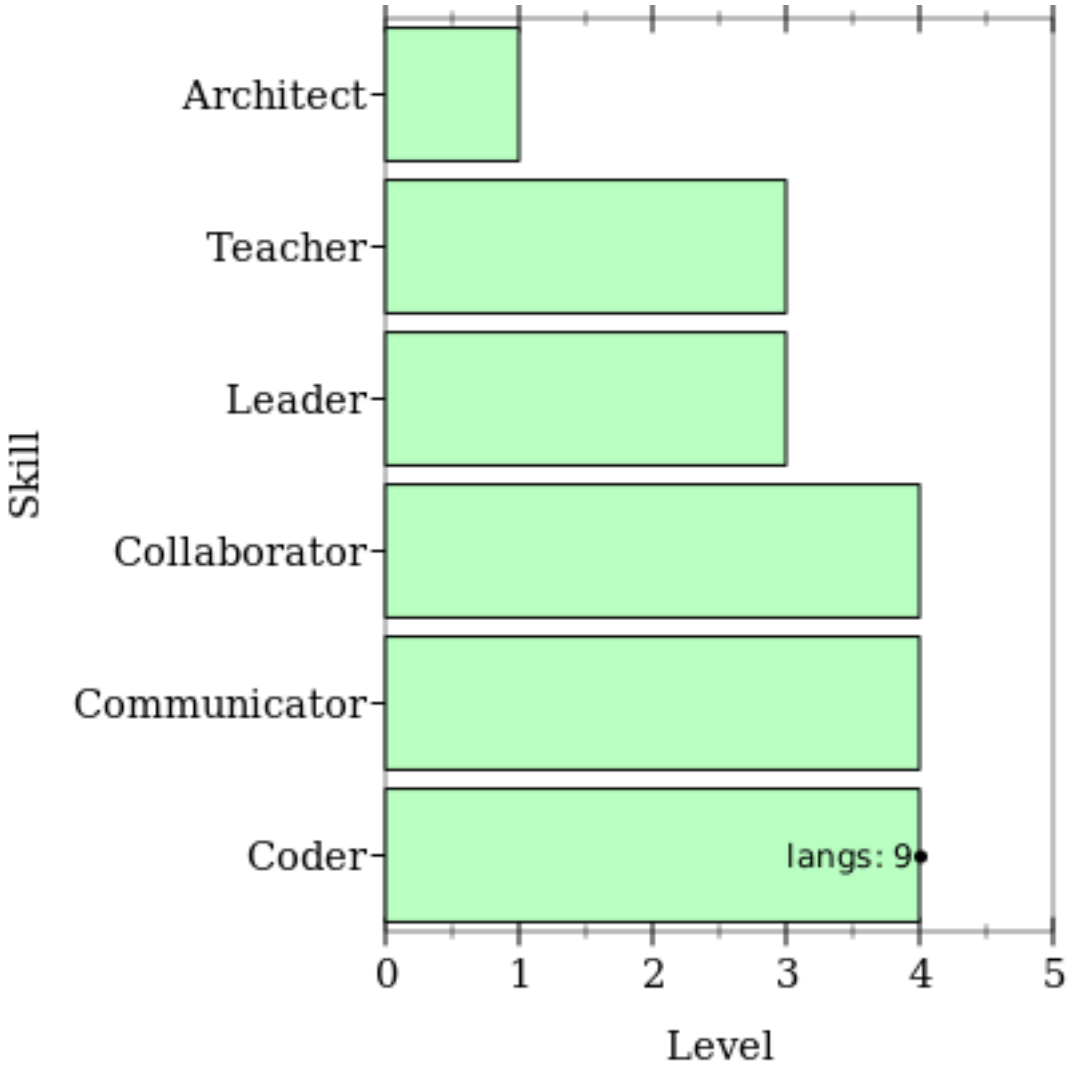
Disclaimer: These are minimum expectations. Many students have been taking the ThoughtSTEM classes and camps for a year – with minimal work outside of class.



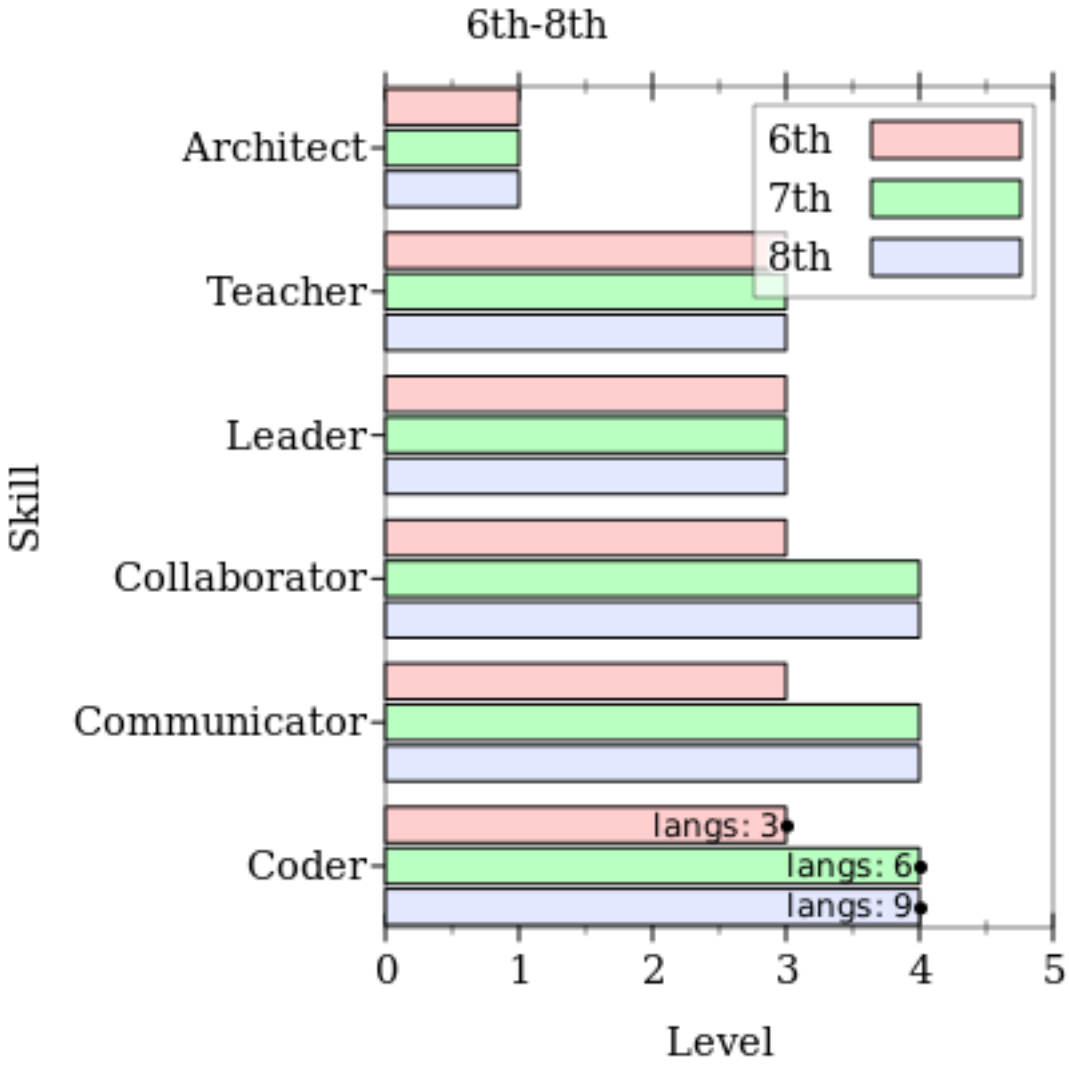
By the end of 7th grade:



By the end of 8th grade:



Comparison across three years:

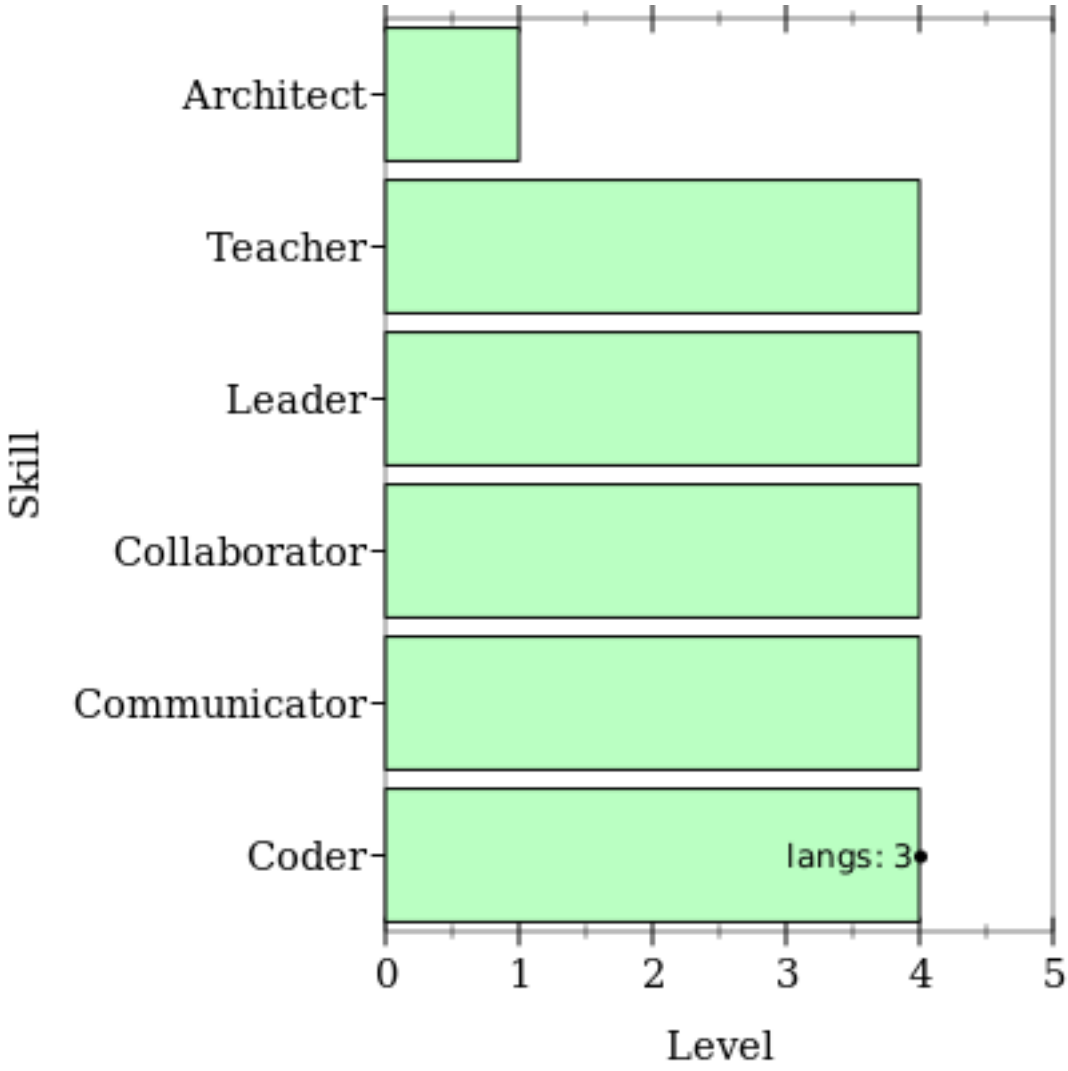


3.6 Expectations: 9th through 12th grade

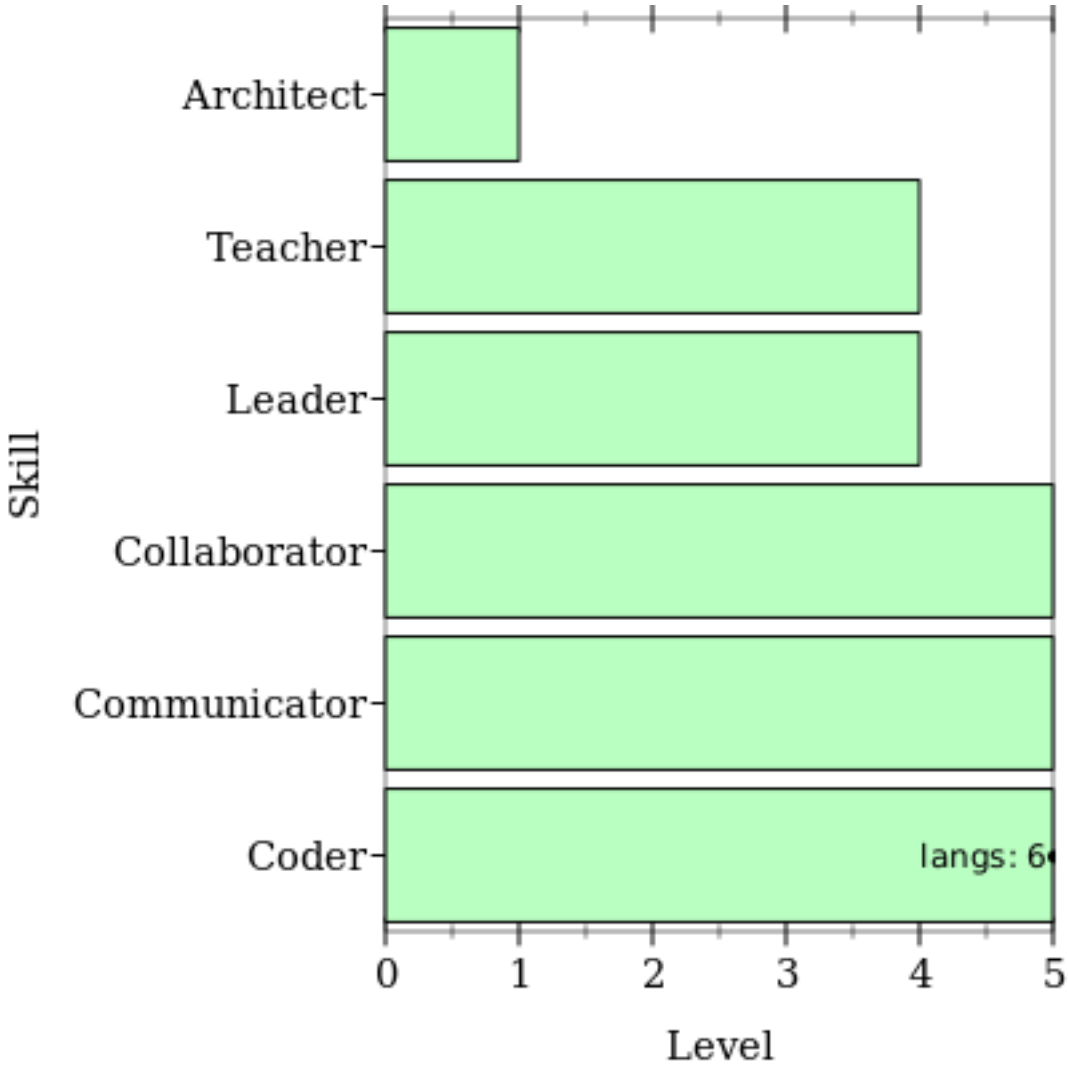
At the 9th through 12th grade levels (compared with younger ages), we expect significantly more growth as **Leaders** and **Teachers**, and moderate growth as an **Architect** . We also expect greater overall growth after 4 years.

By the end of 9th grade:

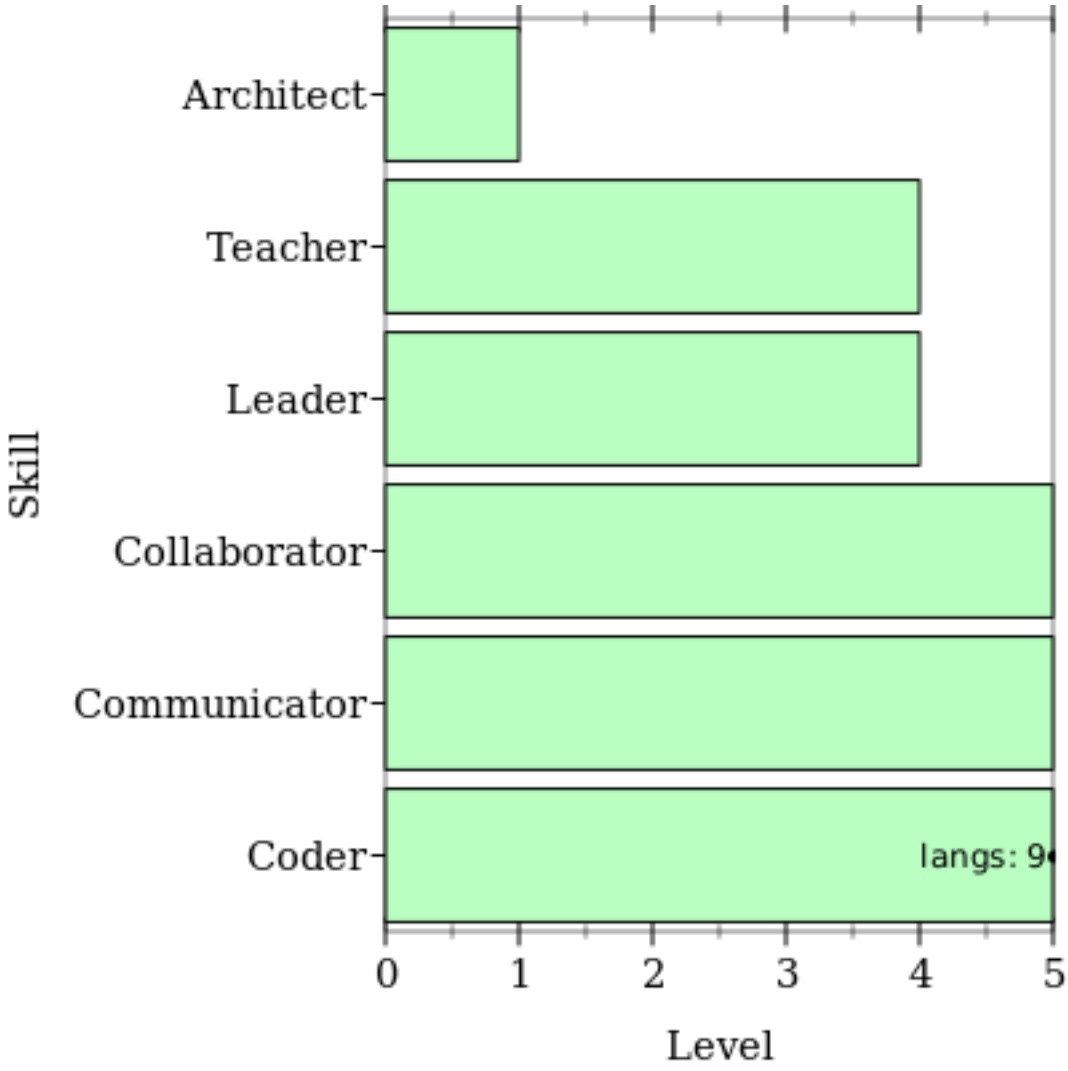
Disclaimer: These estimations of the expectations. Many students have taken the Thought STEM classes discussed here a year – with minimal work outside of class.



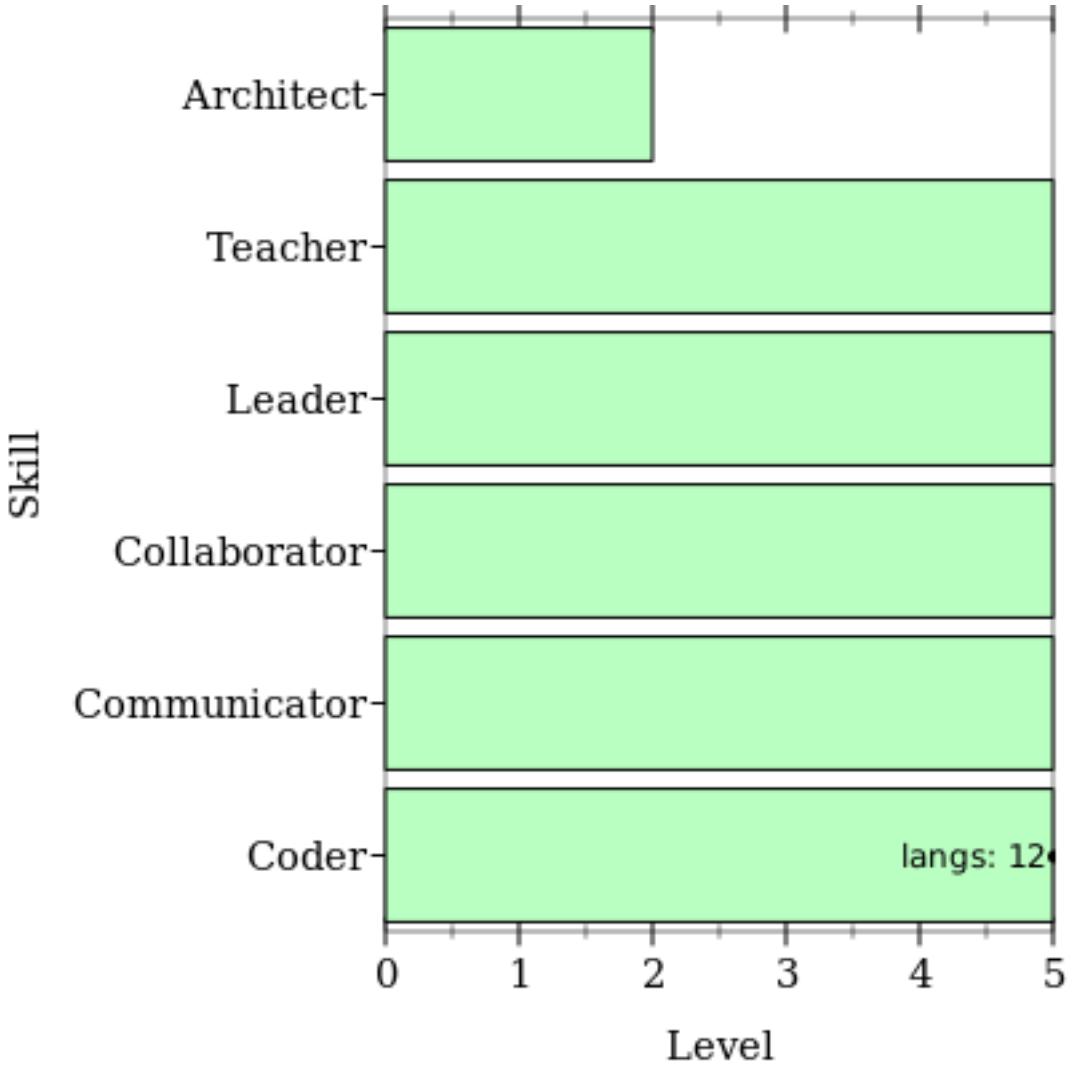
By the end of 10th grade:



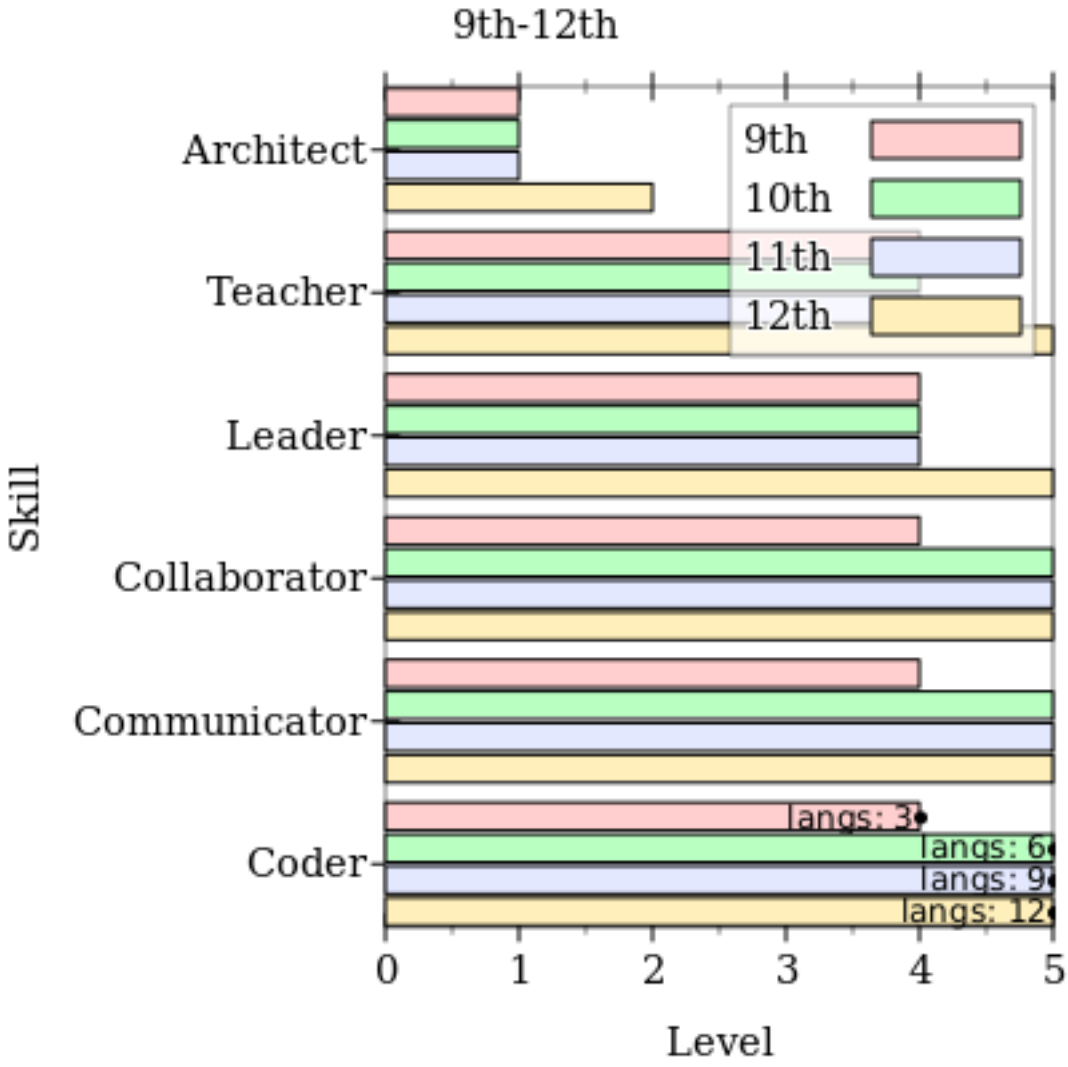
By the end of 11th grade:



By the end of 12th grade:



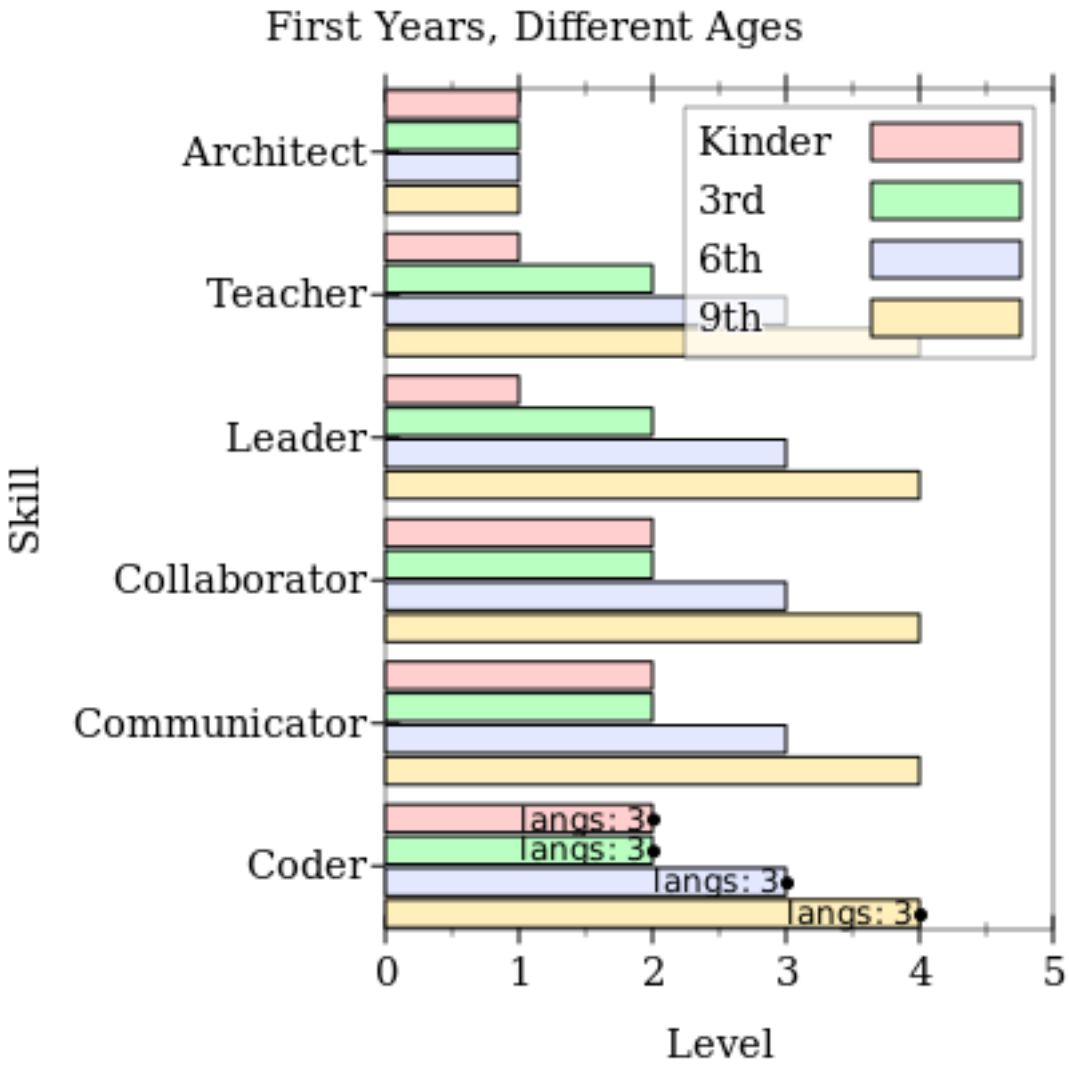
Comparison across all 4 years:



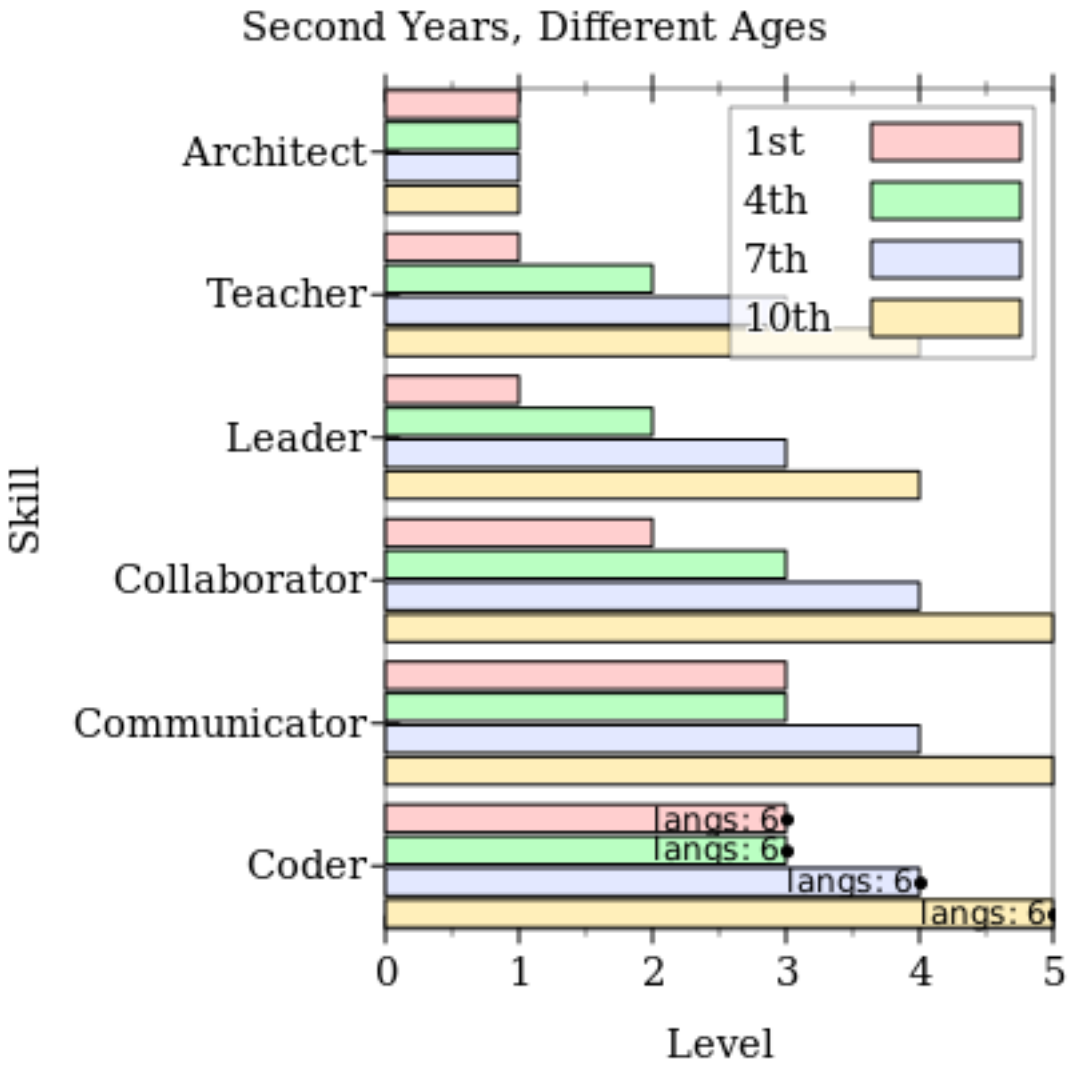
3.7 Comparisons across ages

This section slices the data from the previous sections differently. It shows how our expectations of how students that begin at different ages will progress.

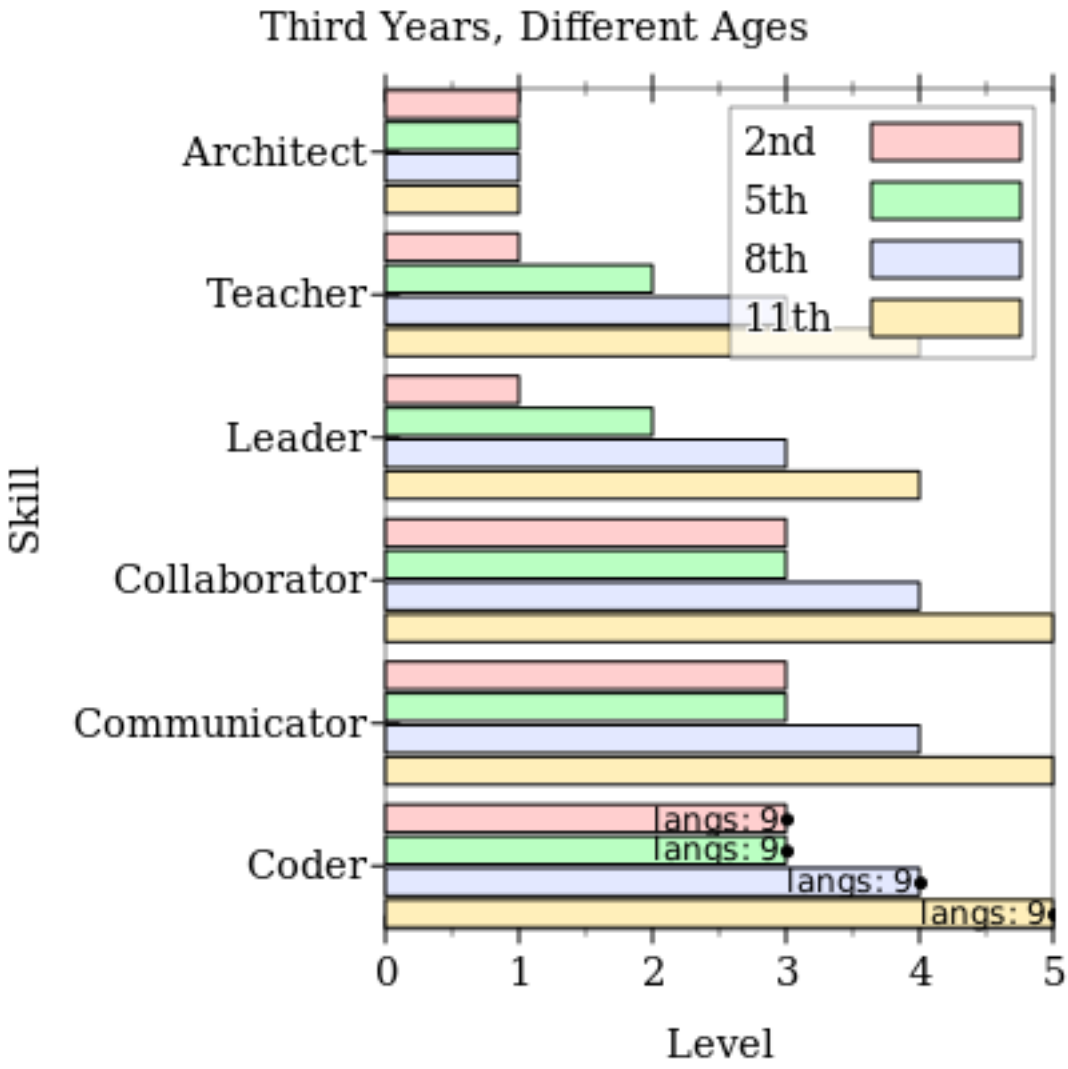
Beginners' skills after 1 year are expected to look as follows, depending on the age of the student:



Beginners' skills after 2 years are expected to look as follows, depending on the age of the student:



Beginners' skills after 3 years are expected to look as follows, depending on the age of the student:


Third Years, Different Ages
Architect
Teacher
Leader
Collaborator
Communicator
Coder
Skill
2nd
5th
8th
11th
0
1
2
3
4
5
Level